NIST Special Publication
800-231

# Bugs Framework (BF)

## *Formalizing Cybersecurity Weaknesses and Vulnerabilities*

Irena Bojanova

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-231

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

# NIST Special Publication 800-231

# 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.SP.800-231

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*

Certain commercial entities, equipment, or materials may be identified in this document in order to describe an experimental procedure or concept adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology (NIST), nor is it intended to imply that the entities, materials, or equipment are necessarily the best available for the purpose.

There may be references in this publication to other publications currently under development by NIST in accordance with its assigned statutory responsibilities. The information in this publication, including concepts and methodologies, may be used by federal agencies even before the completion of such companion publications. Thus, until each publication is completed, current requirements, guidelines, and procedures, where they exist, remain operative. For planning and transition purposes, federal agencies may wish to closely follow the development of these new publications by NIST. Organizations are encouraged to review all draft publications during public comment periods and provide feedback to NIST. Many NIST cybersecurity publications, other than the ones noted above, are available at https://csrc.nist.gov/publications.

**Authority**
This publication has been developed by NIST in accordance with its statutory responsibilities under the Federal Information Security Modernization Act (FISMA) of 2014, 44 U.S.C. § 3551 et seq., Public Law (P.L.) 113-283. NIST is responsible for developing information security standards and guidelines, including minimum requirements for federal information systems, but such standards and guidelines shall not apply to national security systems without the express approval of appropriate federal officials exercising policy authority over such systems. This guideline is consistent with the requirements of the Office of Management and Budget (OMB) Circular A-130. Nothing in this publication should be taken to contradict the standards and guidelines made mandatory and binding on federal agencies by the Secretary of Commerce under statutory authority. Nor should these guidelines be interpreted as altering or superseding the existing authorities of the Secretary of Commerce, Director of the OMB, or any other federal official. This publication may be used by nongovernmental organizations on a voluntary basis and is not subject to copyright in the United States. Attribution would, however, be appreciated by NIST.

**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.SP.800-231

**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 structured 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 by operations of orthogonal software or hardware execution phases, faults – by their input operands, and errors – by their output results. An error either propagates to a fault or is a final error supplying an exploit vector. A BF weakness class is a taxonomic category of a weakness type defined by sets of operations, cause→consequence relations, severity attributes, and code sites. Bugs and faults comprise the causes; errors and final errors – the consequences. A BF weakness is an instance of a taxonomic BF class as a *(cause, operation, consequence)* triple with attributes. A BF vulnerability is a chain of weaknesses linked by causality and consequence→cause propagation, enabling a failure. 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 structured causal taxonomies, bugs models, and vulnerability models. BF's formalism enables a new range of research and development efforts for creation of comprehensively labeled weakness and vulnerability datasets, and diverse multidimensional vulnerability classifications; as well as vulnerability specification generation, bug detection, and vulnerability analysis and remediation. The BF weakness and vulnerability specifications could serve as a formal augmentation to the natural language descriptions of the Common Weakness Enumeration (CWE), the Common Vulnerabilities and Exposures (CVE), and the National Vulnerability Database (NVD).

This Special Publication (SP) presents an overview on the Bugs Framework (BF). Further details will be available in follow-up NIST SP 800-231A-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; CVE; CWE; Cybersecurity; Design; Exploitation; 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 our modern life, economies, and societies, but also form the digital attack surface and may create threat vectors for malicious actors. Cybersecurity of critical infrastructure and supply chains are an increasingly pressing societal challenge. 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.

The current state of the art in Cybersecurity involves use of weakness and vulnerability identifiers and 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 CVE entries (CVEs) with CWE entrees (CWEs). The Known Exploited Vulnerabilities Catalog (KEV) [4] lists publicly exploited CVEs with top priority for remediation.

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 CWE and CVE IDs and natural language descriptions are useful, formal specifications based on structured causal taxonomies would be a valuable augmentation towards applicability is modern security research. The formalism could be used for improving any current imprecise natural language descriptions [5] an especially those with unclear causality and lack explainability [6]. It will help identify gaps and overlaps in CWEs [7] and root causes and sinks in CVEs.

CWE lists CVE examples, but NVD was the first systematic effort toward labeling CVEs with CWEs to allow CVEs classification. Formalizing weaknesses and vulnerabilities will bring this one stage forward as the assignments will be exact, precise, and clear enough for the purposes of the 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 based on structured causal taxonomies 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) [8] is a structured 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 by operations of orthogonal software or hardware execution phases, faults – by their input operands, and errors – by their output results. An error either propagates to a fault or is a final error supplying an ex-

ploit vector. A BF weakness class is a taxonomic category of a weakness type defined by sets of operations, cause→consequence relations, and severity attributes. Bugs and faults comprise the causes; errors and final errors – the consequences. A BF weakness is an instance of a taxonomic BF class as a *(cause, operation, consequence)* triple with attributes. A BF vulnerability is a chain of weaknesses linked by causality and consequence→cause propagation, enabling a failure. 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 structured causal taxonomies, bugs models, and vulnerability models. Analogously to the periodic table BF allows identification/prediction of unencountered yet security weakness types.

BF's strict methodologies for chaining weaknesses underlying a vulnerability and systematic labeling on any level of abstraction can support a new range of cybersecurity research and development efforts for vulnerability specification generation, bug detection, and vulnerability analysis and remediation. It's methodology for backwards identification from a security failure of the bug in software, firmware, or hardware logic, or the initial fault induced by a hardware defect can 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 classifications generation
- 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, and generating comprehensively labeled weakness and vulnerability datasets and vulnerability classifications. Further details will be available in NIST SP 800-231A-I at https://csrc.nist.gov/publications/:

- NIST SP 800-231A BF Security Concepts
- NIST SP 800-231B BF Bugs Models
- NIST SPs 800-231Cx BF _yyy Taxonomies, where _yyy is a BF Class Type ID
- NIST SP 800-231D BF Vulnerability Models
- NIST SP 800-231E BF Formal Language
- NIST SP 800-231F BF Databases, Tools, and APIs
- NIST SP 800-231G BF Weakness and Vulnerability Datasets
- NIST SP 800-231I BF Classifying Vulnerabilities

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. 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. 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) [9, 10] 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.

The CVE was initiated in 1999 [11] to address the problem of having "no common naming convention and no common enumeration of the vulnerabilities in disparate databases" [12]. The CWE followed in 2006 to address "the issue of categorizing software weaknesses" and establishing "acceptable definitions and descriptions of these common weaknesses"; and "support for hardware weaknesses" was added in 2020 [13]. Both CVE and CWE adopted a one-dimensional list (enumeration) approach of organizing the entries by unique IDs with natural language descriptions; CWE also added tree based abstractions. They are regularly refined and new publicly disclosed vulnerabilities, weakness types, and related content are added [13]. Past classification efforts as "the development of the periodic table of elements and the identification of animals" [12] 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 ..." [12]. "... The [security] 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 ..." [12].

CWE and CVE are widely used by the security community at large. CWE provides descriptions and information on modes of introduction, possible mitigations, detection methods, and demonstrative examples. CVE provides descriptions and references to reports, proof or concept, and code. 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. Focusing on the CWE and CVE descriptions, many are not sufficient, accurate, and precise enough [5, 14, 15] and have unclear causality [16–18]. CWE has gaps and overlaps in coverage [16–18], some of which have been recently pointed by the Hardware CWE Special Interest Group (HW CWE

SIG) [19]. Many CVEs list as root cause the final error at the sink, instead of the bug or hardware defect induced fault that starts the chain of weaknesses underlying a vulnerability. Some CVEs list the root cause, but do not list the final error at the sink. Many CVEs do not list the entire chain of weaknesses underlying the vulnerability. These CWE and CVE challenges propagate also to NVD and KEV, and may lead to imprecise or wrong CWE to CVE assignments by NVD or third-party analysts. (see Fig. 1) Additionally, CWE and CVE do not exhibit strict methodologies for tracking the weaknesses underlying a vulnerability, for systematic comprehensive vulnerability labeling, and for backwards root cause identification from a security failure. There are no tools to aid users in the creation and visualization of weakness and vulnerability 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.

The imprecise descriptions and luck of explainability make CWE and CVE difficult to use in modern cybersecurity research [6]. Augmenting their natural language descriptions[1] with unambiguous formal specifications will make them more suitable to be used as comprehensively labeled datasets for training ML and AI models either [20].

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. [17]

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. [17] 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 root cause.

[1]Focus of this work is the weaknesses and vulnerabilities descriptions – the need of formal causal specifications that enhance understanding and support methodologies for finding and fixing the bugs.

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 [7].

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. It could be utilized to augment NVD and from there KEV, as well as directly CWE (to the extend possible) and CVE with unambiguous weakness and vulnerability BF specifications.

## 3. BF Approach

The Bugs Framework (BF) is a formal multi-dimensional classification. Its approach is different from the exhaustive, ID based list approach exhibited by enumerations. BF classes are identifiable by the orthogonal (non-overlapping) sets of operations by which they organize bugs and faults. They allow expressing a specific weakness via its cause (operation bug or operand fault) and consequence (error or final error operation result), as well as severity attributes, and code sites.

First ideas about formalizing software bugs [14] and a "Periodic Table" of bugs [21] started forming in 2014-2015. The first BF classes [15, 22, 23], however, were rather naive, as there were no ideas on what should be causes and consequences organized by; on formal weaknesses and vulnerabilities causation, propagation, convergence, and chaining; nor on failure classes. The classes from [15, 22–24] had to be reclassified and redeveloped. The BF systematic approach and methodology for organizing security bugs and faults were discovered via basic research in 2019 and the follow up publications [16–18, 20, 25] started demonstrating their application. BF, BFCWE, and BFCVE tools started being created in 2020 while iteratively developing BF taxonomies, models, specifications structure, and causation and propagation rules. The BF formal grammar syntax and semantics rules and the BF formal language were created in 2023.

A BF weakness class is a taxonomic representation 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 their input operands and output results. An operation defines what the software or hardware performs on an appropriate level of abstraction. Operations or operands improperness define the causes – bugs and faults, correspondingly. A consequence is the erroneous result of the operation over the operands – an error that propagates to a fault or is a final error supplying an exploit vector towards a failure. The attributes describe the operations and the operands and help understand the severity of the bug or fault causing the weakness. The sites point to syntactic places in code that should be checked for that class of bugs or faults.

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 comprises:

- Strict *definitions* of bug, fault, error, final error, weakness, vulnerability, and 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, and context-free bugs and faults *taxonomies* as BF weakness classes, and exploit vector *taxonomies* as BF failure classes.
- A formal *vulnerability state model* of chains of bug and faults states leading to a failure(s).
- A formal *vulnerability specification model* of chained BF weakness class instances leading to a BF failure class instance.
- 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.
- Multidimensional security *vulnerability classifications*.

BF's taxonomies are structured via cause-operation-consequence relations, complete – no gaps in coverage, orthogonal by operation, and context-free – language and domain independent. In more details:

*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 severity related attributes.

*Complete* means 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. This assures the BF weakness types have clear causality and 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.

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 final error. BF's formalism guarantees unique precise descriptions of weaknesses (including CWE) and vulnerabilities (including CVE); and complete orthogonal weakness types coverage (without gaps and overlaps). It enables the creation of comprehensively labeled weakness and vulnerability datasets and formal security vulnerability classifications.

## 3.1. BF Security Weakness

BF models a *security weakness* as an improper state and its transition (see Fig. 2). 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.

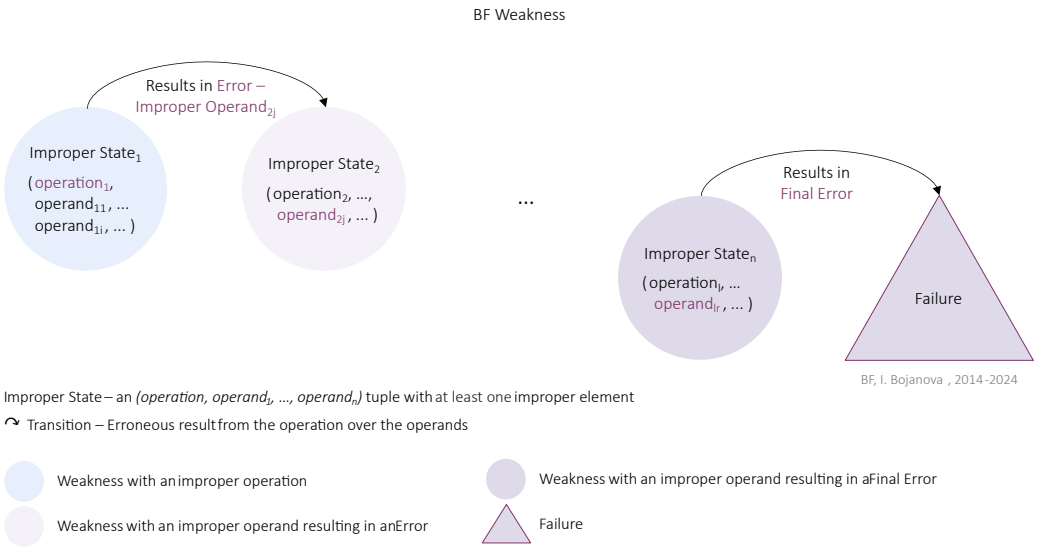



**Fig. 2.** BF Weakness. An improper state defined as an $(operation, operand_1, \cdots, operand_n)$ tuple with at least one improper element, and a transition defined by the erroneous result from the operation over the operands.

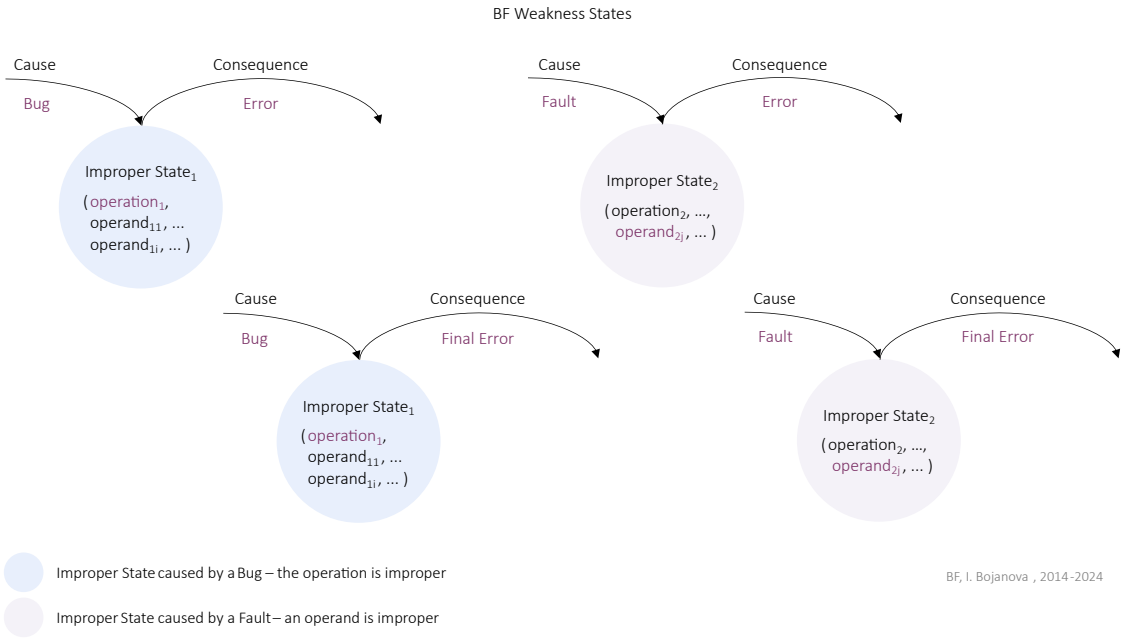



**Fig. 3.** BF Weakness States. A BF weakness is a *(cause, operation, consequence)* triple – formally, one of the possible *(bug/fault, operation, error/final error)* causation triples.

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.

Figure 3 presents the possible BF weakness states – a bug state is depicted in blue, a fault state is depicted in purple. A weakness 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 an error – that propagates to another fault, or a security final error – an exploitable or undefined system behavior.

A bug is a software, firmware, or hardware code or specification defect. Specification is about operation's metadata or algorithm. A bug (the improper operation) may result from a hardware defect or resurface from system configuration or environment change. A hardware defect may be due due to overheating, electromagnetic fields, wear and tear, etc.

A fault is a name, data, type, address, or size error. Name is about a resolved or bound object, function, data type, or namespace; data, type, address, and size are about an object. A fault (the improper operand) could result from a bug or induced by a hardware defect.

A BF weakness is as a *(cause, operation, consequence)* triple, which formally is a *(bug, operation, error)*, *(fault, operation, error)*, *(bug, operation, final error)*, and *(fault, operation, final error)* causation triples. It is of a bug weakness type or of a fault weakness type. A bug informs the operation is improper; a fault informs about an improper operand.

### 3.2. BF Security Vulnerability

BF models a *security vulnerability* as a chain of improper states (see Fig. 4) propagating towards a failure as an erroneous result (*error*) from one state becomes an improper operand (*fault*) for a next state, until a *final error* that can be exploited towards a security *failure* is reached.

The initial state – depicted in blue – is usually caused by a *bug* – a code or specification defect within the operation. A weakness chain may also start from a hardware defect induced *fault* of an operand. A propagation state – in light purple – is caused by at least one fault, an ill-formed operand. The final state – in dark purple – results in a *final error* (a undefined system behavior) and leads to a *failure* (a violation of a system security requirement). The final error supplies an exploit vector. It usually directly relates to a CWE; however, the opposite is not true – there are CWEs that cover initial or propagation weakness states.

An *error* is the result of an improper state from an operation over its operands. It becomes an improper operand, *fault*, 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 error, leading to a failure.

The initial *bug* state is of an improper operation over proper operands; it is the state with defect in the operation; the bug must be fixed to resolve the vulnerability. A *fault* state is of a proper operation over at least one improper operand; it is a state with a defect in an operand, which if fixed would only mitigate the vulnerability. An improper operation or improper operand may result from a hardware (HW) defect.

Vulnerabilities may also converge at their final states and chain via exploitation resulting faults towards a security failure.

While the root cause of a vulnerability may be attributed to different stages of software design life cycle (SDLC) or hardware design life cycle (HDLC) – e.g., the design stage, the focus of BF is on the possible eventual bugs in executable operations and the possible faults of their operands. BF's ultimate goal is a mechanism for a bug and faults detection from an occurred failure, as fixing the bug would resolve the vulnerability; fixing a fault – would remediate it. //xxx IB: Should I reiterate on this is some other setions? I do not want to get comments like "A bug is a design flaw!"xxx//

### 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

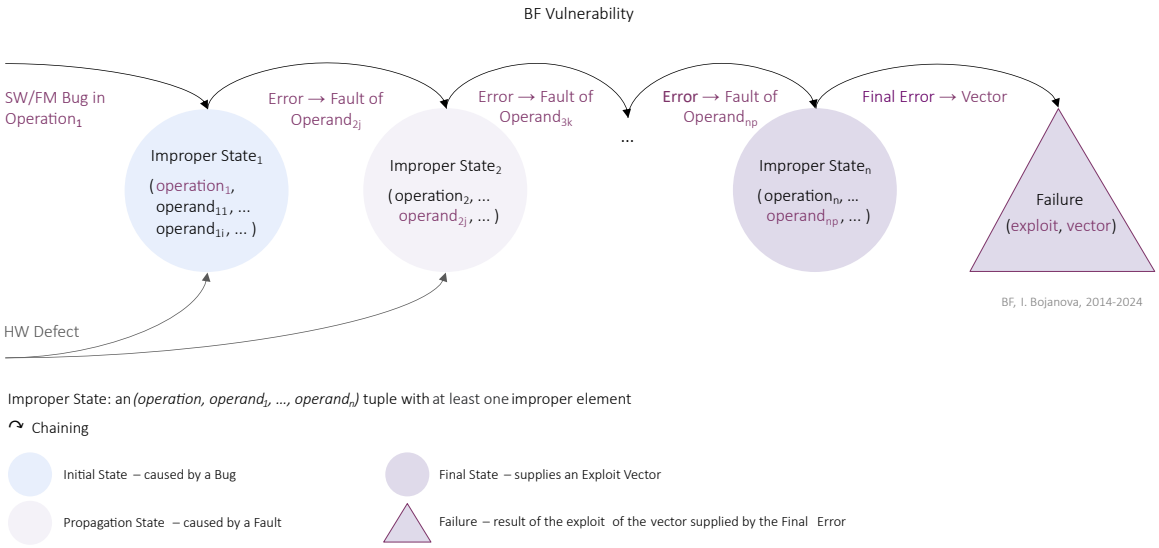



**Fig. 4.** BF Vulnerability. A chain of weaknesses as improper states. Starts with a SW/FM bug, propagates via *error→fault* transitions towards a final error, enabling a security failure. An improper operation (the bug) or operand (a fault) may result from a HW defect. SW stands for software; FM – for firmware (including microcode); HW – for hardware.

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. 7.2), 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.

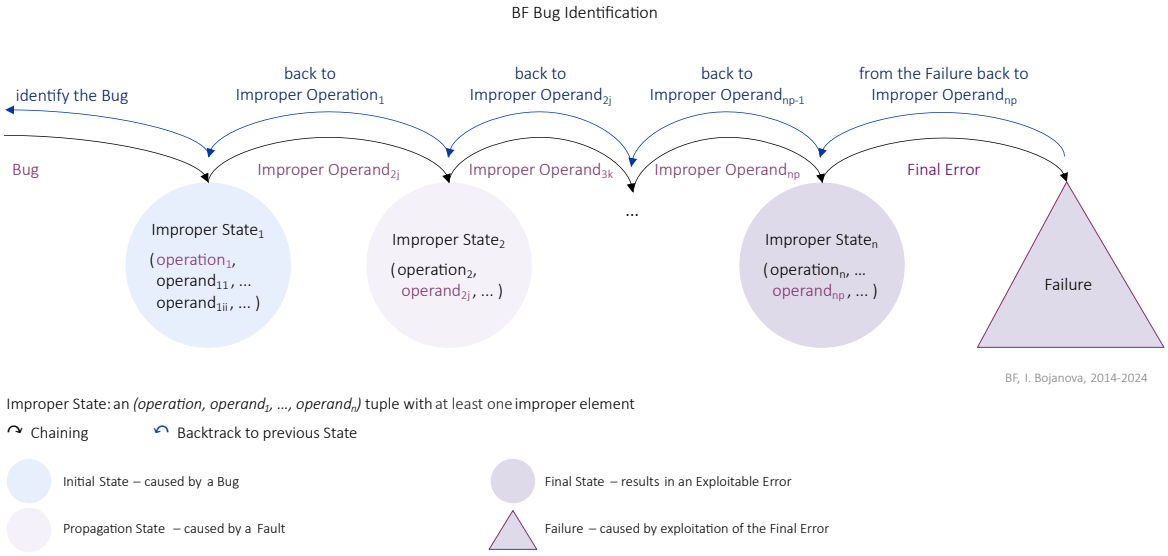



**Fig. 5.** BF Bug Identification. Going backwards from a Failure through Faults to the Bug.

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

## 4. BF Security Concepts

A *security bug* type or a *fault* type relates to a distinct phase of execution – of software, firmware (incl. microcode), or hardware circuit logic – defined by a set of operations and their input operands and output results.

Examples of operations are dereference, write, or deallocate memory, but also data type related condition evaluation, as well as encryption of plain text. Examples of operands are object address, data type, and object size.

BF defines the concepts of bug, fault, error, final error, weakness, vulnerability, and failure in the context of cybersecurity as follows. They provide the level of granularity needed to understand the causation within a weakness, as well as the causation and propagation between weaknesses and between vulnerabilities.

- A *security bug* is a code or specification defect (an operation defect) in software, firmware, or hardware logic – proper operands over an improper operation. Specification is about operation's metadata or algorithm. A bug may result from a hardware defect or resurface from system configuration or environment change. A hardware defect may be due due to overheating, electromagnetic fields, wear and tear, etc.

  Examples of code bugs are missing code (e.g., an entire operation is missing) or erroneous code (e.g., a wrong operator is used). Examples of specification bugs are use of wrong algorithm for encryption or use of under-restrictive safelist for input validation. While, an operation may run perfectly on a 64-bit operating system (OS) environment, it may exhibit a security bug on a 32-bit platform – e.g., *int* instead of *uint* declaration propagating through wrap around (integer overflow) towards a 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. A fault could result from a bug or induced by a hardware defect.

Examples of faults are wrong value, dangling pointer, wrong type, or weak cryptographic key. Wrong value could result from missing validation or from an erroneous calculation, but also – from bit flips or signal disruption from electromagnetic interference (EMI), voltage or clock glitches, photon injection, overheating, and physical hardware damage.

- An *error* is a result from an operation with a bug or a faulty operand that propagates to a faulty operand of another operation.
- A *security final error* is an exploitable or undefined system behavior – supplies an exploit vector.

  Examples of final errors are integer overflow, query injection, buffer overflow.
- A *security weakness* is a *(bug, operation, error)*, *(fault, operation, error)*, *(bug, operation, final error)*, or *(fault, operation, final error)* causation triple.
- A *security vulnerability* is a causal chain of weaknesses that starts with a bug or with a hardware defect induced fault, propagates through errors that become faults, and ends with a final error. The first weakness relates to its root cause. The last weakness relates to its sink.
- A *security failure* is a violation of a system security requirement – results from an exploit via a vector supplied by a final vulnerability error.

  Examples of failures are information exposure (IEX) – confidentiality loss, data tempering (TPR) – integrity loss, denial of service (DoS) – availability loss, arbitrary code execution (ACE) – everything could be lost.

The BF security concepts definitions are contextually visualized on Fig. 6.

Fixing the bug or starting fault of a vulnerability – will resolve it, as well as any other vulnerability with the same root cause. Fixing a fault or the final error at the sink may only mitigate the vulnerability.

Occasionally, for an exploit to be harmful, several vulnerabilities must converge at their final errors. Fixing the bug or the starting fault of at least one of the chains would avoid the failure.

An exploit of a vulnerability may result in a fault starting a new faults-only vulnerability. Fixing the bug or the starting fault the first vulnerability will resolve the entire chain of vulnerabilities.

For details on the BF concepts definitions refer NIST SP 800-231A BF Security Concepts.

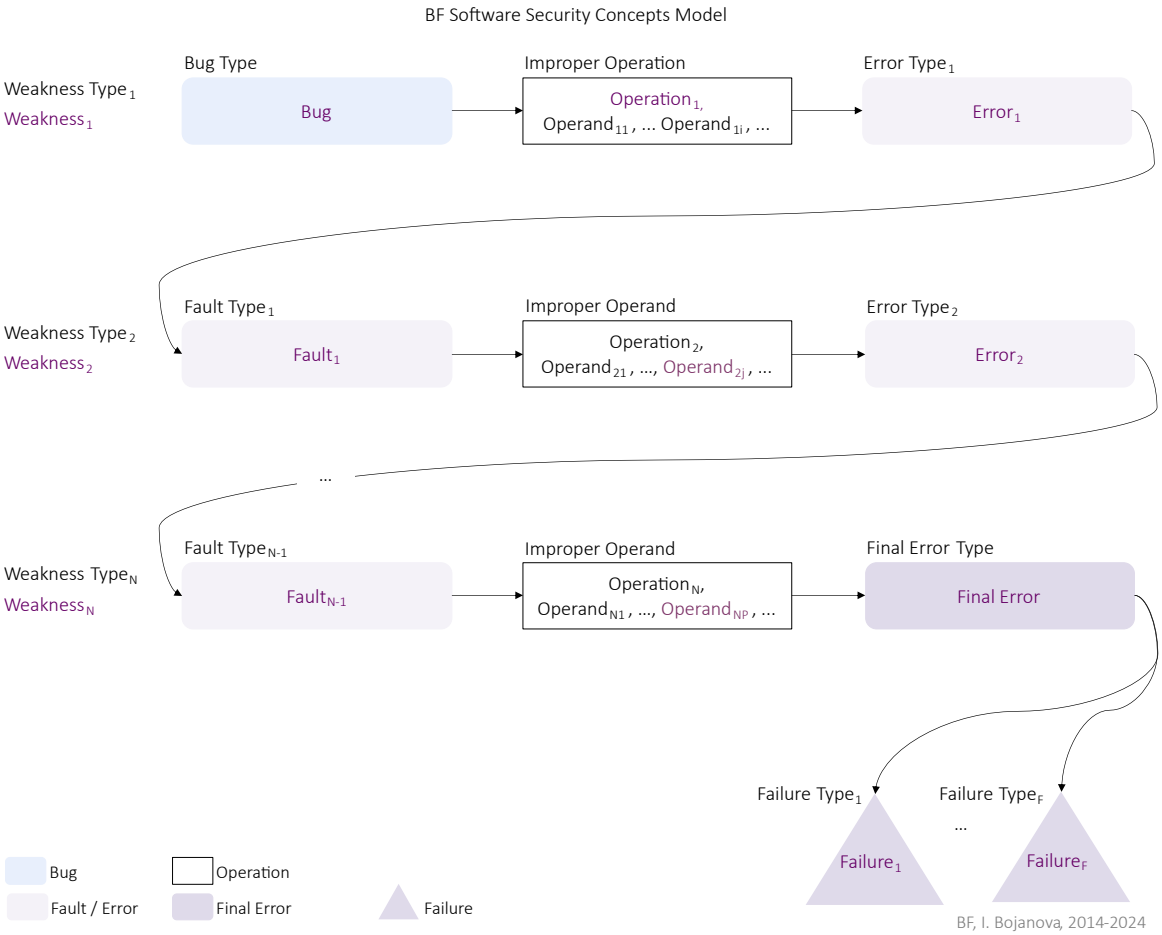



**Fig. 6.** BF Security Concepts. Vulnerability built by Weakness built by Bug, Fault, Error, and Final Error towards Failure.

## 5. BF Bugs Models

The BF bugs and faults landscape covers the operations in software, firmware (including microcode), and hardware execution phases on appropriate levels of abstraction. The software and firmware operations relate to code in 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 software and firmware. Some phases of execution may be only on application level (e.g., input/output check), others may cover deeper 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 with a bug or a hardware defect induced faulty operand that propagated through faulty operands of other operations until a final error supplying an exploit vector is reached.

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. 7) 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*.

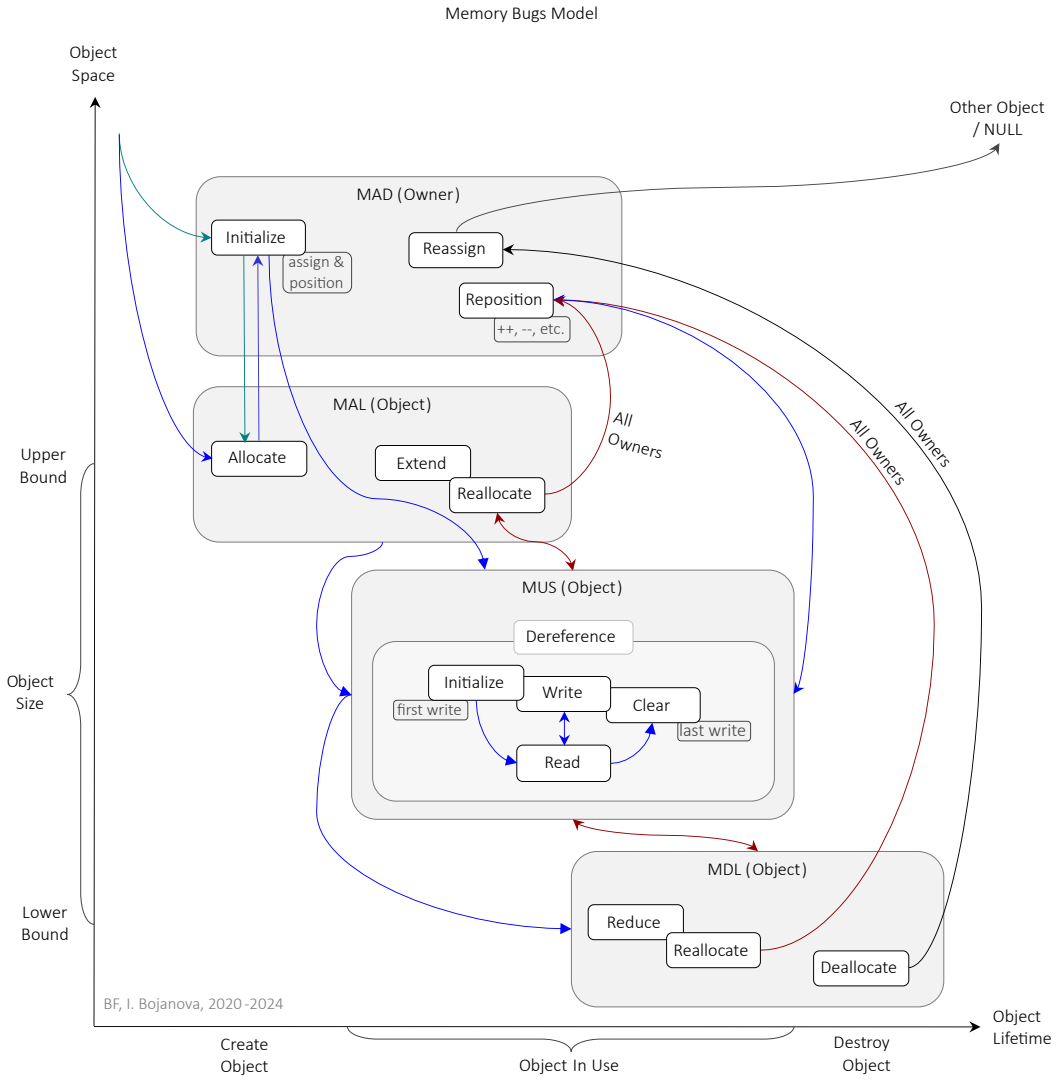



**Fig. 7.** BF Memory Bugs Model. Shows memory related software and firmware execution phases, non-overlapping by operations, where bugs or faults could happen, and operations flow.

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 (e.g. floats safety), input/output safety, and memory safety.

As another example, Data Type bugs (see Fig. 8) 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*.

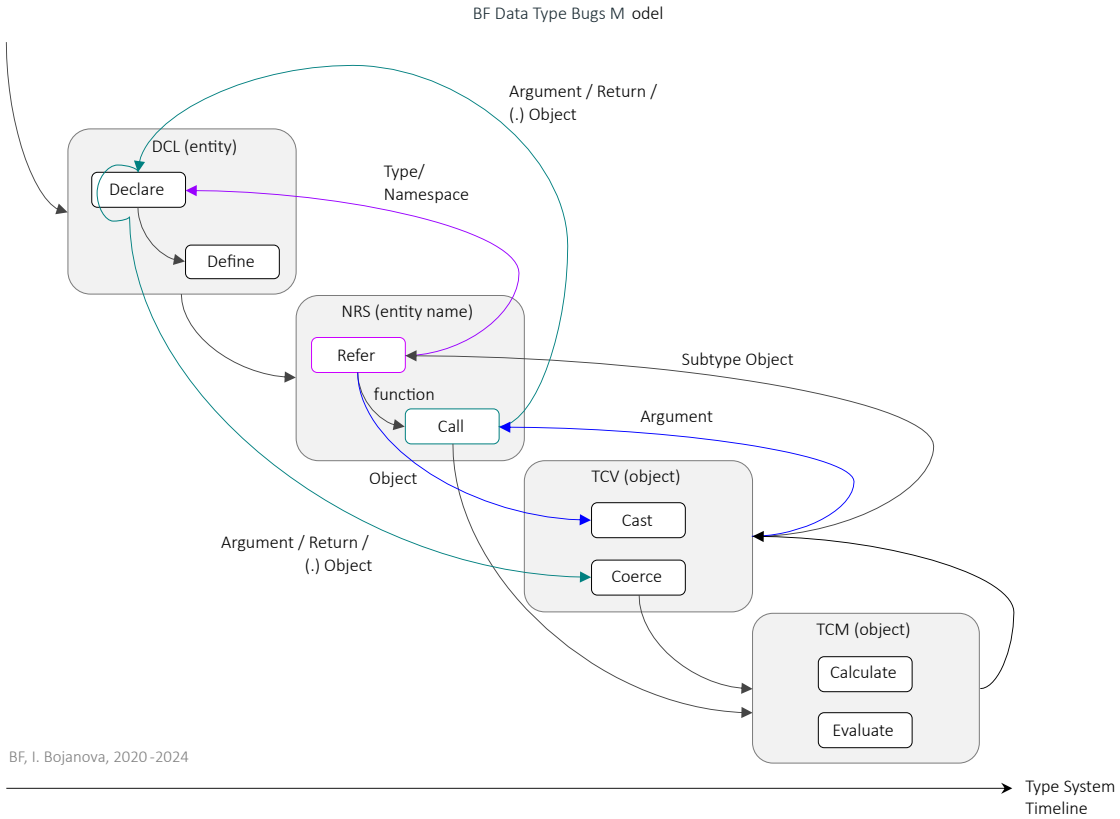



**Fig. 8.** BF Data Type Bugs Model. Shows data type related software and firmware execution phases, non-overlapping by operations, where bugs or faults could happen, and operations flow.

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. 8 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 converted to another data type and used in computations as an argument or as a return of a called function, or to call a member function. A passed in argument is expected to be of the declared parameter data type and the passed out result is expected to be of the return

data type. Otherwise, casting (explicit conversion) is expected before or at the end of the call (see the blue arrows flow), or the value will get coerced (implicitly converted) to the parameter data type or the return data type, correspondingly (see the green arrows flow). Note that the green arrows flow is only about coerced passed in/out objects – it starts only from NRS Call, it never starts from DCL Declare.

As another example, the possible flow between memory related operations is depicted in Fig. 7 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 800-231C BF Bugs Models.

## 6. BF Taxonomy

The BF taxonomy comprises weakness and failure categories (see its XML representation structure on Fig. 9, query it via the BF API).

The BF Weakness category comprises of BF Weakness class types – e.g.,:

- *Data Type (_DAT)* class type – Bugs/Faults supplying type compute exploit vectors.
- *Input/Output Check (_INP)* class type – Bugs/Faults supplying injection exploit vectors.
- *Memory Corruption/Disclosure (_MEM)* class type – Bugs/Faults supplying memory corruption/disclosure exploit vectors.

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

```
<!--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. 9.** BF Taxonomy Structure – weakness category with bugs/faults–operations related class types, and a failure category with vector–exploit related classes.

## 6.1. BF Weakness Classes

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

fines sets of possible bugs and faults as causes for the operations of a specific phase over their operands to result in errors and final errors as consequences. As by definition an error propagates to a fault, the set of errors across classes is the same to the set of faults across classes. The set of final errors across the classes is the same as the set of exploit vectors allowing failures. A BF weakness class defines also severity attributes and code sites.

A BF weakness class type encompasses weakness classes of closely related execution phases. For example, the BF _DAT class type comprises 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.

As other examples, the BF _INP class type encompasses the Data Validation (DVL) and Data Verification (DVR) bugs classes. The BF _MEM class type groups the Memory Addressing (MAD), Memory Management (MMN), Memory Use (MUS) classes.

Figure 10 depicts the BF Type Conversion (TCV) class [18], organizing security bugs by the *Cast* and *Coerce* operations, and faults by their Name, Data, and Type operands to cause errors such as *Wrong Type*, *Wrong Value*, *Flipped Sign*, *Truncated Value*, *Distorted Value*, *Rounded Value*, *Flipped Sign* and *Truncated Value*. Figure 11 depicts the BF Data Validation (DVL) class [17], organizing bugs/faults by *Validate* and *Sanitize* operations and their *Data* operand, causing *Invalid Data* error or final injection errors such as *Query Injection* and *Source Code Injection*. Figure 12 depicts the BF Memory Use (MUS) class [16] , organizing bugs/faults by memory specific operations and their operands, causing *Uninitialized Object* error or final errors such as *Use After Deallocate* (e.g., *Use After Free* and *Use After Return*) and *Buffer Overflow*.

Each BF class strictly defines the value taxons (e.g., see the purple terms on Fig. 10, 11, and 12) for class and operations; and type and value taxons for causes (as bugs or faults), consequences (as errors or final errors), and operation and operand attributes. 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.

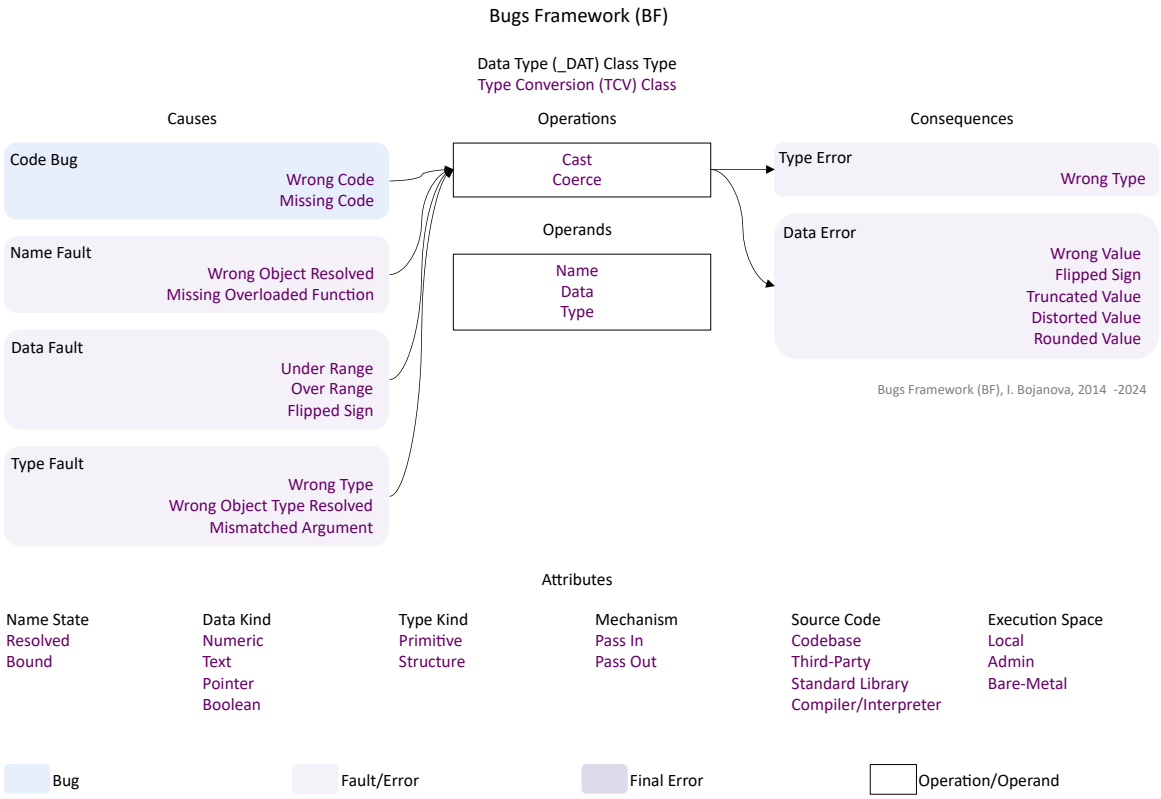



**Fig. 10.** BF Type Conversion (TCV) class of BF Data Type (_DAT) class type.

As further examples, the definitions of the taxon values *Data Validation (DVL)* class, *Missing Code* bug, *Corrupted Data* fault, *Validate* operation, and *Query Injection* final error (see them on Fig. 12) are as follows – some may include specific examples.

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

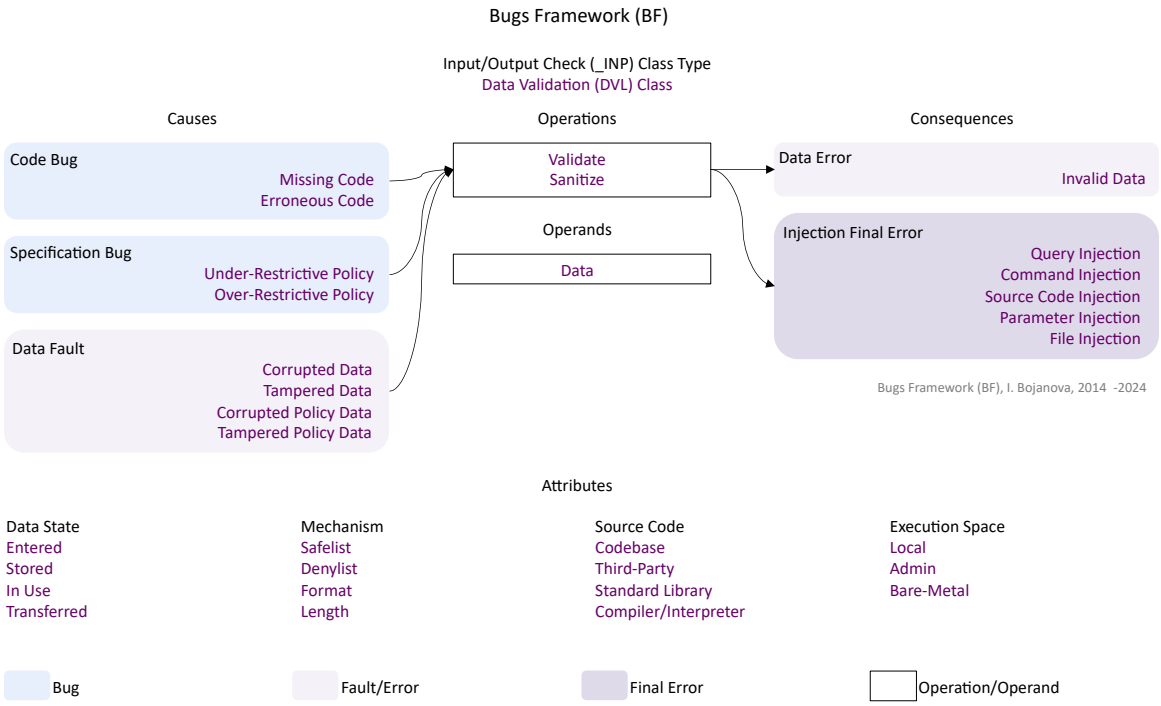



**Fig. 11.** BF Data Validation (DVL) class of BF Input/Output (_INP) class type.

*Size* cause, *Wrong Write* cause, and *Buffer Overflow* final error (see them on Fig. 12) 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. 9) of the BF taxonomy. They are visualized also as tooltips of the BF Tool [26] and under the BFCVE specifications on the BF Website [8].

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

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, attributes, and possibly sites – selected from the values of the causes, operations, consequences, attributes, and sites taxons of that class. The operation binds the

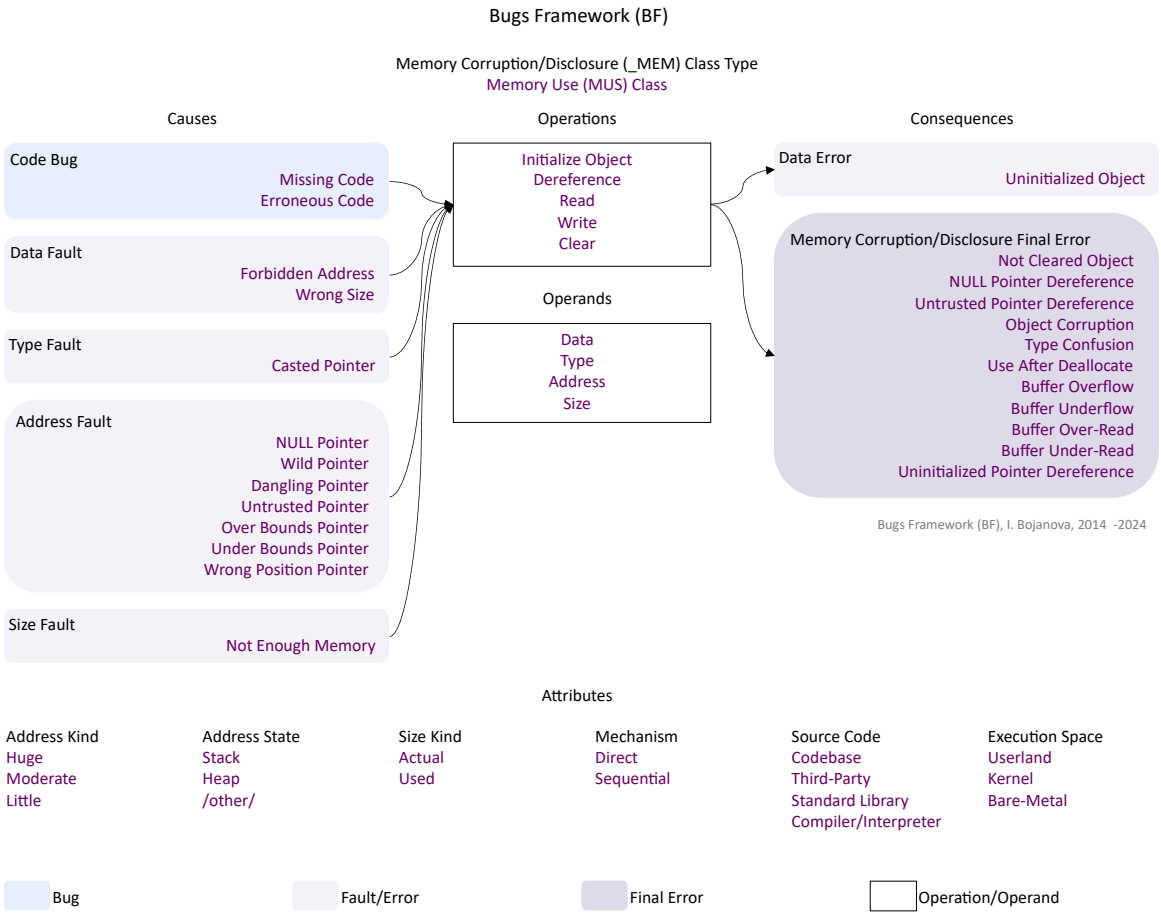



**Fig. 12.** BF Memory Use (MUS) class of BF Memory Corruption/Disclosure (_MEM) class type.

*cause→consequence* relation – e.g., *Missing Validate* may lead to the final error known as *Query Injection*; and *Write* via *Over Bounds Pointer* leads to the final error known as *Buffer Overflow*.

For the complete set of developed BF class types, classes, taxon definitions, and semantic matrices refer NIST SPs 800-231Cx BF _yyy Taxonomies, where _yyy is a BF Class Type ID. For the complete BF formal language lexis refer NIST SP 800-231E BF Formal Language.

## 6.2. BF Failure Class

A BF failure class defines the possible exploit vectors for that failure and results that may start new vulnerabilities. The exploit vectors correspond to final errors of BF weakness classes. Information exposure or data tempering failure may result in a fault starting a new vulnerability. The BF _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.
- *Data Tempering (TPR)* – Unauthorized modification or destruction of information – integrity loss.

### 6.3. BF Methodology

The methodology for developing BF weakness classes is as follows (see Fig. 13).

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 weakness 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. 12) correspond to the related memory addressing and memory use software or firmware execution phase. They are also of the BF Memory Corruption/Disclosure (_MEM) [16] class type.
2. Operations: Identify all 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. 12).
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. 7). 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. 7) 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* values for the weakness triples (see Fig. 3). For example, DVL has two bug values and two specification values (see Fig. 11).
5. Fault Causes: Identify all input operands for the operations – these would define the possible fault types. For example, see the *Data Fault*, *Type Fault*, *Address Fault*, *Size Fault*, *Data Error* types listed for MUS (see Fig. 12). Identify the possible operand errors – these would define *fault* values for the weakness triples (see Fig. 3). For

example, there are 11 *fault* values for MUS on (see Fig. 12).

6. Error Consequences: Identify all output result 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* values for the weakness triples (see Fig. 3). For example, there are 6 *error* values for TCV on (see Fig. 10) and one – for DVL (see Fig. 11) and MUS(see Fig. 12).
7. final error Consequences: Identify all output result errors from the operations that do not propagate as causes for other weaknesses – these would define the possible *final error* consequence values for the weakness triples (see Fig. 3). For example MUS has one final error type – Memory Corruption/Disclosure; and 11 final error values (see Fig. 12).
8. Operation Attributes: Identify specific descriptive values for the following operation attribute types.
    - *Execution Space* operation attribute type – Shows where the buggy/faulty operation is running or with what privilege level.
    - *Mechanism* operation attribute type – Shows how the buggy/faulty operation is performed.
    - *Source Code* operation attribute type – Shows where the buggy/faulty operation is in the program – in what kind of software.
9. Operand Attributes: Identify specific descriptive values for each operand type *Kind* and *State* attribute types.
    - *Address Kind* operand attribute type – Shows what the accessed outside object's bounds memory is.
    - *Address State* operand attribute type – State operand attribute type – Shows where the address is in the memory layout.
    - *Data Kind* operand attribute type – Shows what the data value is.
    - *Data State* operand attribute type operand attribute – Shows where the data come from.
    - *Name Kind* operand attribute type – Shows what the entity with this name is.
    - *Name State* operand attribute type – Shows at what stage the entity name is.
    - *Size Kind* operand attribute type – Shows what the limit for traversal of the object is.
    - *Type Kind* operand attribute type – Shows what the data type composition is.

10. Sites: Identify possible sites in code for such bugs/faults - a step applicable mainly for low level bugs.

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

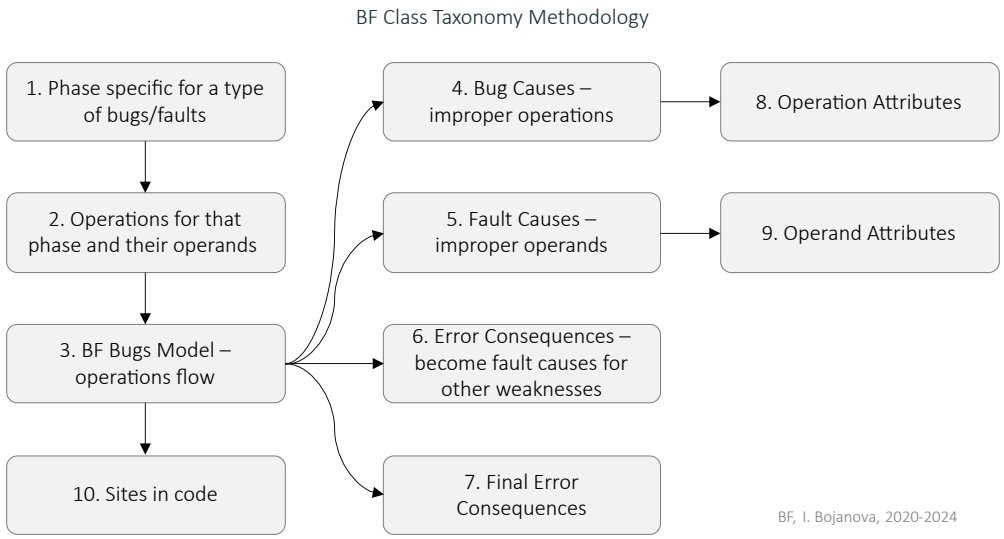



**Fig. 13.** BF Methodology for developing a BF weakness class.

## 7. 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.

### 7.1. BF Vulnerability State Model

BF models a *vulnerability* (see Fig. 14) 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* usually corresponds to a weakness caused by a *bug* in the operation, resulting in an Error. It may also correspond to a weakness caused by a faulty operand induced by a hardware defect. 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 a final error. The Failure is a violation of a security requirement caused by an exploit leveraging a vector created by the final error.

Fixing the Bug – the operation implementation or the specification defect – will resolve the vulnerability; fixing a fault would mitigated it. Fixing a bug may relate to unaccounted system configuration or environment. Fixing a fault may involve fixing a hardware defect.

Occasionally, for an exploit to be harmful, several vulnerabilities must converge (depicted with $\oplus$) at their final errors. Fixing the bug or the starting fault of at least one of the chains would avoid the failure.

An exploit of a vulnerability may result in a fault starting a new vulnerability of only fault type weaknesses. Fixing the bug or the hardware defect staring the first vulnerability will resolve the entire chain of vulnerabilities.

### 7.2. BF Vulnerability Specification Model

BF models a *vulnerability specification* (see Fig. 15) 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. 6) 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 error. More specifically, as matrices of meaningful *(bug, operation, error)*, *(fault, operation, error)*, *(bug, operation, final error)*, or *(fault, operation, final 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.

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*$\rightarrow$*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*$\rightarrow$*cause*. For example, *(Wrong Type, Coerce, Flipped Sign)* may propagate to *(Wrong Argument, Evaluate, Under Range)* as Evaluate may follow Coerce (see Fig. 8), the within weakness causations and *Flipped Sign*$\rightarrow$*Wrong Argument* transition are valid.

Causation between vulnerabilities is by a valid *exploit*$\rightarrow$*operation* transitions – the result from an exploit starts a new faults-only vulnerability. Propagation between vulnerabilities is by the *exploit result* type matching to a start $fault$ type. For example, exposed private keys may become the fault starting a new vulnerability.

For simplicity Fig. 15 does not show vulnerability convergence and chaining, as they correspond directly to the transitions in the Vulnerability State Model (see 14).

## 8. 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

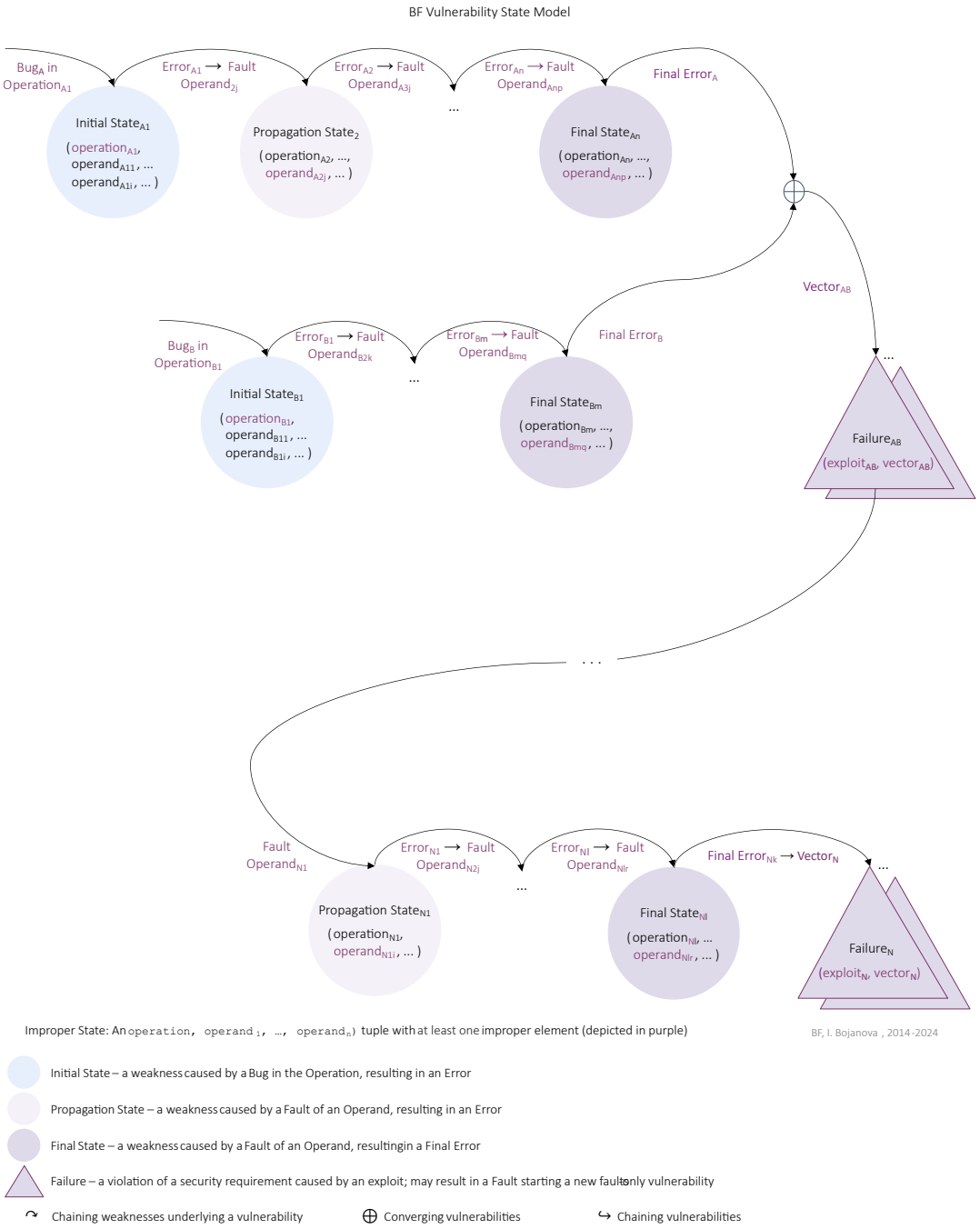



**Fig. 14.** BF Vulnerability State Model. Starts with a bug or a fault from a hardware defect, propagates via errors becoming faults, leading to a failure. Occasionally, vulnerabilities must converge for an exploit to be harmful. Could propagate to other faults-only-vulnerabilities.

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

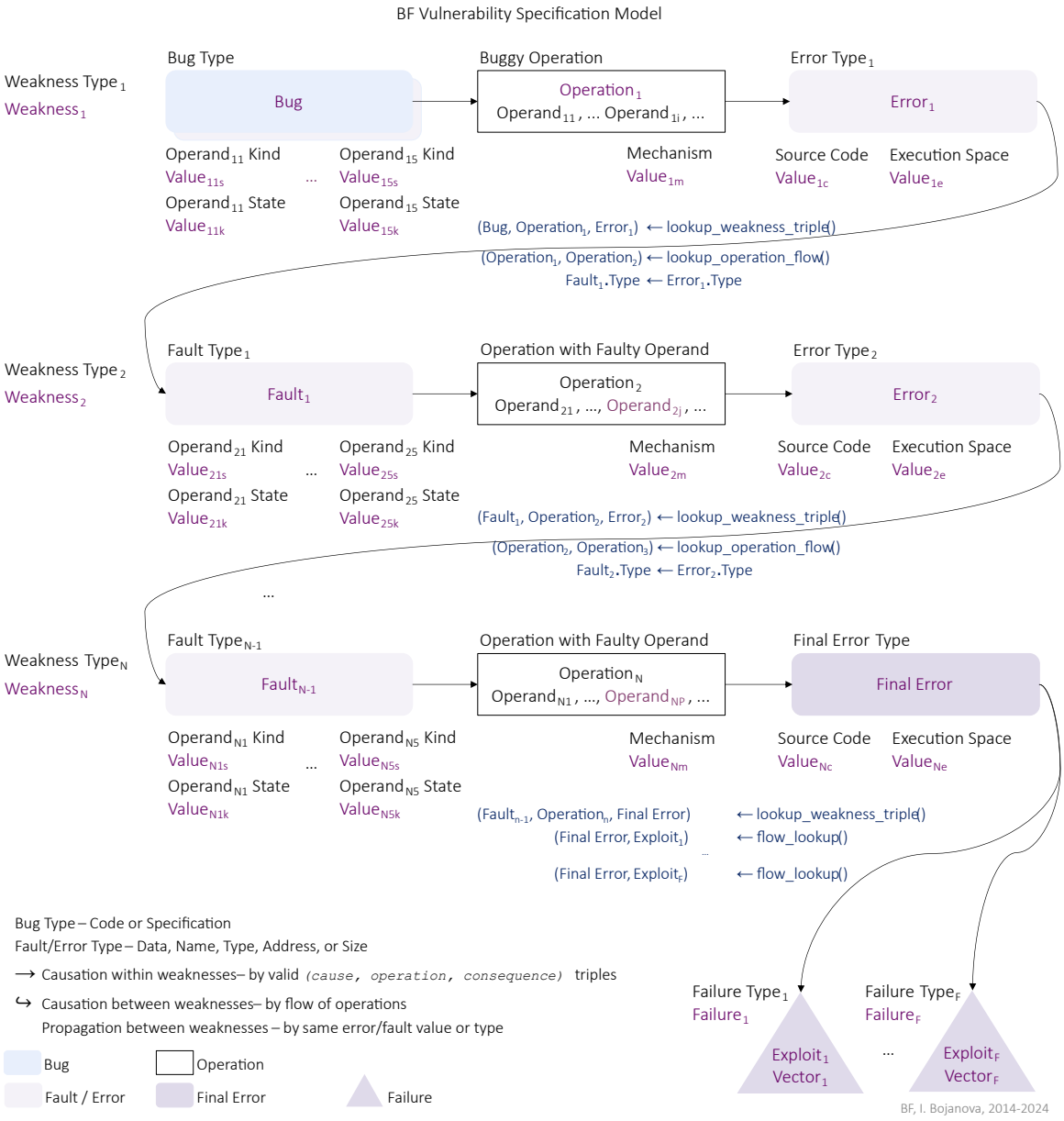



**Fig. 15.** BF Vulnerability Specification Model. Reflects BF's concepts definitions, taxonomies' within weakness causation rules, bugs models' between weaknesses causation rules, and state model's propagation rules.

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 structured causal taxonomies (see for example Fig. 12), bugs models (see, for example, Fig. 7), and vulnerability models (see Fig. 14 and 15). *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)

$$\Sigma = \left\{ \alpha \mid \alpha \in \Sigma Taxon \cup \Sigma Symbol \right\}$$

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

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

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

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

    , where:

    $\left( V \cup \Sigma \right)^*$ 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.

## 8.1. 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 type and classes of bug, fault, final error, operation attribute, operand attribute, failure, 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.

$$\Sigma = \{\Sigma Taxon, \Sigma Symbol\} \tag{3}$$

, where

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

, 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:

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

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

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

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

For complete BF formal grammar lexis and taxons definitions refer NIST SP 800-231B BF Formal Language.

## 8.2. 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. 15) (including the BF state model converging and chaining – see Fig. 14). 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. 15) 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 propagation 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.

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

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

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).

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

$$\begin{aligned}
Vulnerability &::= Bug_Fault\ Operation\ OperAttrs_Error_FinalError \\
Bug_Fault &::= Bug \\
&\quad | \ Fault \\
OperAttrs_Error_FinalError &::= OperationAttribute\ OperAttrs_Error_FinalError \\
&\quad | \ Error\ Fault\ OprndAttrs_Operation \\
&\quad | \ FinalError \\
OprndAttrs_Operation &::= OperandAttribute\ OprndAttrs_Operation \\
&\quad | \ Operation\ OperAttrs_Error_FinalError \\
Converge_Failure &::= \oplus\ Vulnerability\ Converge_Failure \\
&\quad | \ Vector\ Exploit\ NextVulner_Failure \\
NextVulner_Failure &::= Fault\ OprndAttrs_Operation \\
&\quad | \ Failure\ \varepsilon
\end{aligned}$$

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 800-231E BF Formal Language.

## 8.3. 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. 15) (including the BF state model converging and chaining – see Fig. 14).

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)

$$
\begin{aligned}
&\mathit{SyntaxRules}: \\
&S ::= \mathit{Vulnerability}\ \mathit{Converge\_Failure} \\
&\mathit{Vulnerability} ::= \mathit{Bug\_Fault}\ \mathit{Operation}\ \mathit{OperAttrs\_Error\_FinalError} \\
&\mathit{Bug\_Fault} ::= \mathit{Bug} \\
&\quad \mid \mathit{Fault} \\
&\mathit{OperAttrs\_Error\_FinalError} ::= \mathit{OperationAttribute}\ \mathit{OperAttrs\_Error\_FinalError} \\
&\quad \mid \mathit{Error}\ \mathit{Fault}_1\ \mathit{OprndAttrs\_Operation} \\
&\quad \mid \mathit{FinalError} \\
&\mathit{OprndAttrs\_Operation} ::= \mathit{OperandAttribute}\ \mathit{OprndAttrs\_Operation} \\
&\quad \mid \mathit{Operation}_k\ \mathit{OperAttrs\_Error\_FinalError} \\
&\mathit{Converge\_Failure} ::= \oplus\ \mathit{Vulnerability}\ \mathit{Converge\_Failure} \\
&\quad \mid \mathit{Vector}\ \mathit{Exploit}\ \mathit{NextVulner\_Failure} \\
&\mathit{NextVulner\_Failure} ::= \mathit{Fault}_2\ \mathit{OprndAttrs\_Operation} \\
&\quad \mid \mathit{Failure}\ \varepsilon
\end{aligned}
$$

$$
\begin{aligned}
SemanticRules: & \\
(Bug,\ Operation_1,\ Error) &\leftarrow lookup\_weakness\_triple() \\
(Bug,\ Operation_1,\ FinalError) &\leftarrow lookup\_weakness\_triple() \\
(Fault_1,\ Operation_k,\ Error),\ k>1 &\leftarrow lookup\_weakness\_triple() \\
(Fault_1,\ Operation_k,\ FinalError),\ k>1 &\leftarrow lookup\_weakness\_triple() \\
(Operation_1, ...Operation_k),\ k>1 &\leftarrow lookup\_operation\_flow() \\
Fault_1 \leftarrow if\ (Fault_1.ClassType &== Error.ClassType)\ then\ Error \\
Predicates: & \\
Fault_1.Type &== Error.Type \\
Vector.Type &== FinalError.Type \\
Fault_2.Type &== ExploitResult.Type
\end{aligned}
$$

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 [27]).

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 *Vector* to store the final error types (e.g., *Injection*, *Access*, *Type Compute*, *Memory Corruption/Disclosure*, ...).

For the complete BF LL(1) formal CFG NIST SP 800-231E BF Formal Language.

## 9. 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.

### 9.1. BF Databases

The BFDB database hosts the BF data. The BF taxonomy structure and the BF formal language rules are organized via relational (see diagram at BFDB Diagram) and graph databases, and data interchange formats such as XML and JSON (query it via the BF API). The databases contain the types, names, and definitions of the BF taxons, and their relationships within the taxonomy, as well as the BF weaknesses and vulnerabilities causation and propagation mattresses.

The VulDB mashup database defines and organizes additional data for querying BF towards CWE, CVE, NVD, and GitHub [28], KEV [4], Software Assurance Reference Dataset (SARD) [29], and Exploit Prediction Scoring System (EPSS) [30] repositories.

For details on the BF databases refer NIST SP 800-231F BF Databases, Tools, and APIs.

## 9.2. BF Tools

The BFCWE tool and the BFCVE tool [31] tool facilitate generation of formal weakness and vulnerability specifications. The BF tool guides the creation of complete BF vulnerability specifications.

### 9.2.1. BFCWE Tool

The BFCWE tool [32] facilitates the creation of CWE-to-BF (CWE2BF) mappings by weakness operation, error, and final 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 computer science and security 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)*, *(bug, operation, final error)*, or *(fault, operation, final error)* weakness triples (see [18], [17], [16], and [7]).

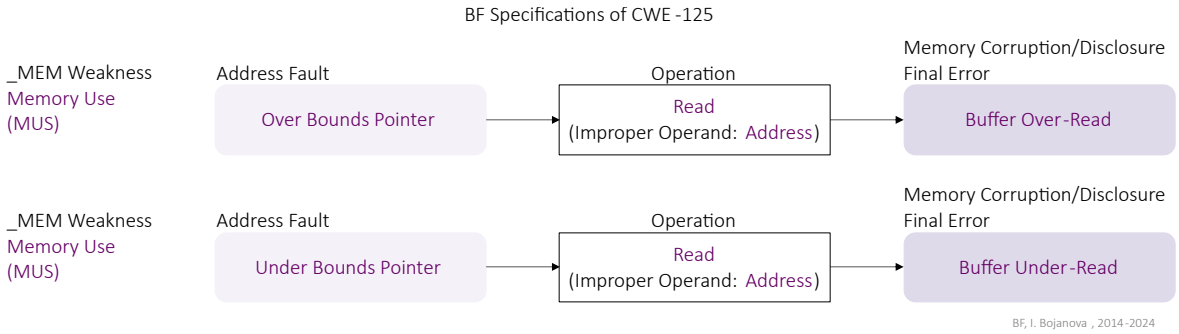
The BFCWE tool generates BFCWE formal specifications as entries of the BFCWE security weakness types dataset. It also generates di-graphs for enhanced understanding of the CWE2BF mappings (by operation, error, final error, and complete weakness triples) with parent-child CWE relations, and of the BFCWE formal specifications.

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 [7] for _MEM).

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




| **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.

specifications. Although, a CWE should be about a single weakness, the descriptions of some CWEs also reveal possible causing chains of weakness triples [7].

For details on the BFCWE tool refer NIST SP 800-231F BF Databases, Tools, and APIs.

### 9.2.2. BFCVE Tool

The BFCVE tool [31] generates possible chains of weaknesses for a vulnerability by identified failure and final error or possibly entire final 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) [18] weakness triples by CWE, more than 4600 – to BF Input/Output Check (_INP)

[17] weakness triples, and more than 3970 CVEs – to BF Memory Corruption/Disclosure (_MEM) [16] 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 [33] for which final weaknesses map to BF Memory Corruption/Disclosure (_MEM [16]) 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 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 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 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 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 [25]. The CWE2BF mappings for CWE-125 restricts to two the final 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 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 [16].

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 error is the first node, and a bug is the last node in each path. The only options for the weakness causing the final 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/Correct, 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, Incomplete 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 800-231F BF Databases, Tools, and APIs.

### 9.2.3. 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 security vulnerability as a chain of underlying weaknesses. A 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 error is reached, causing a security failure. The causation within a weakness is by a meaningful *(cause, operation, consequence)* triple. The causation and

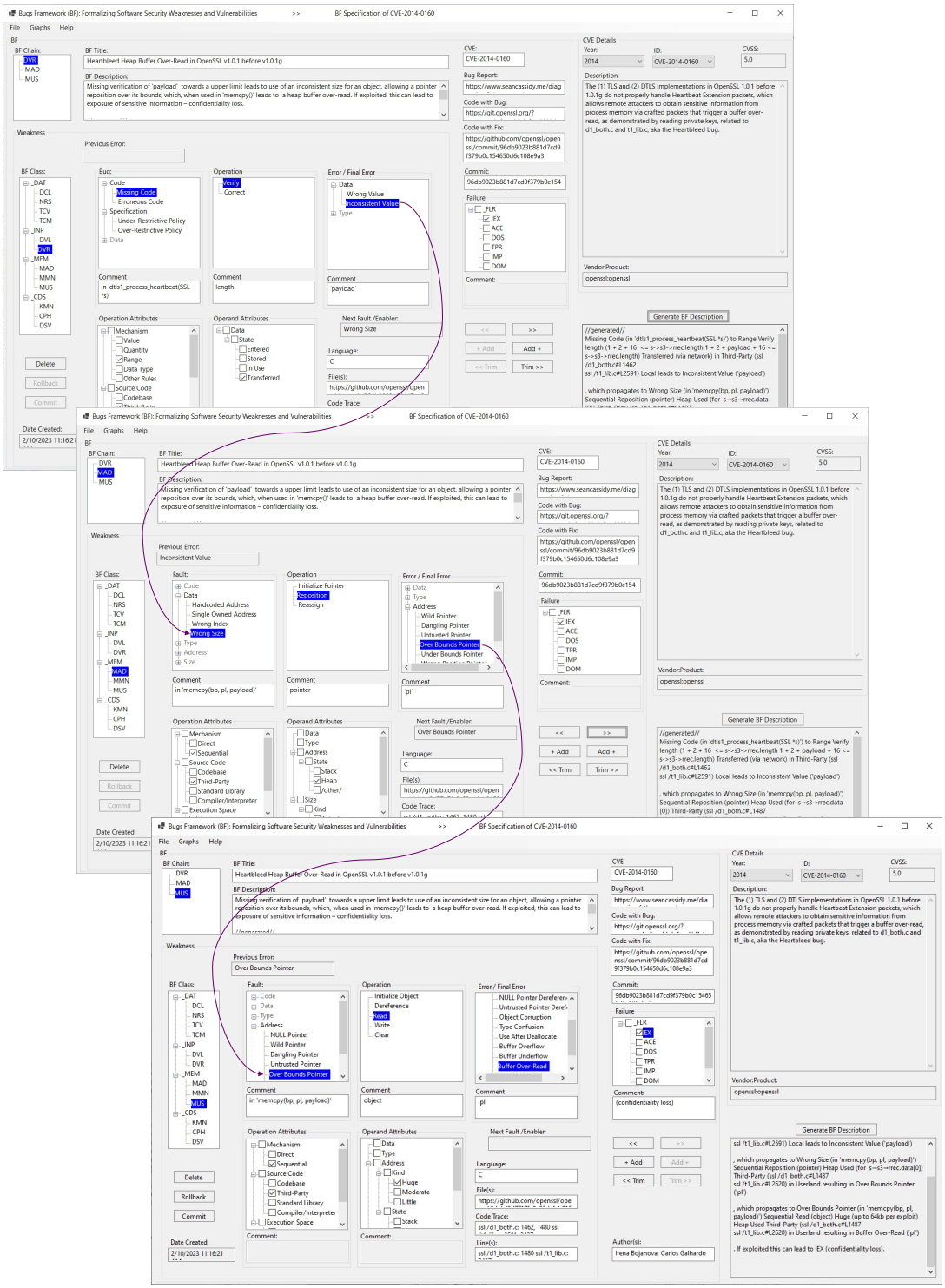


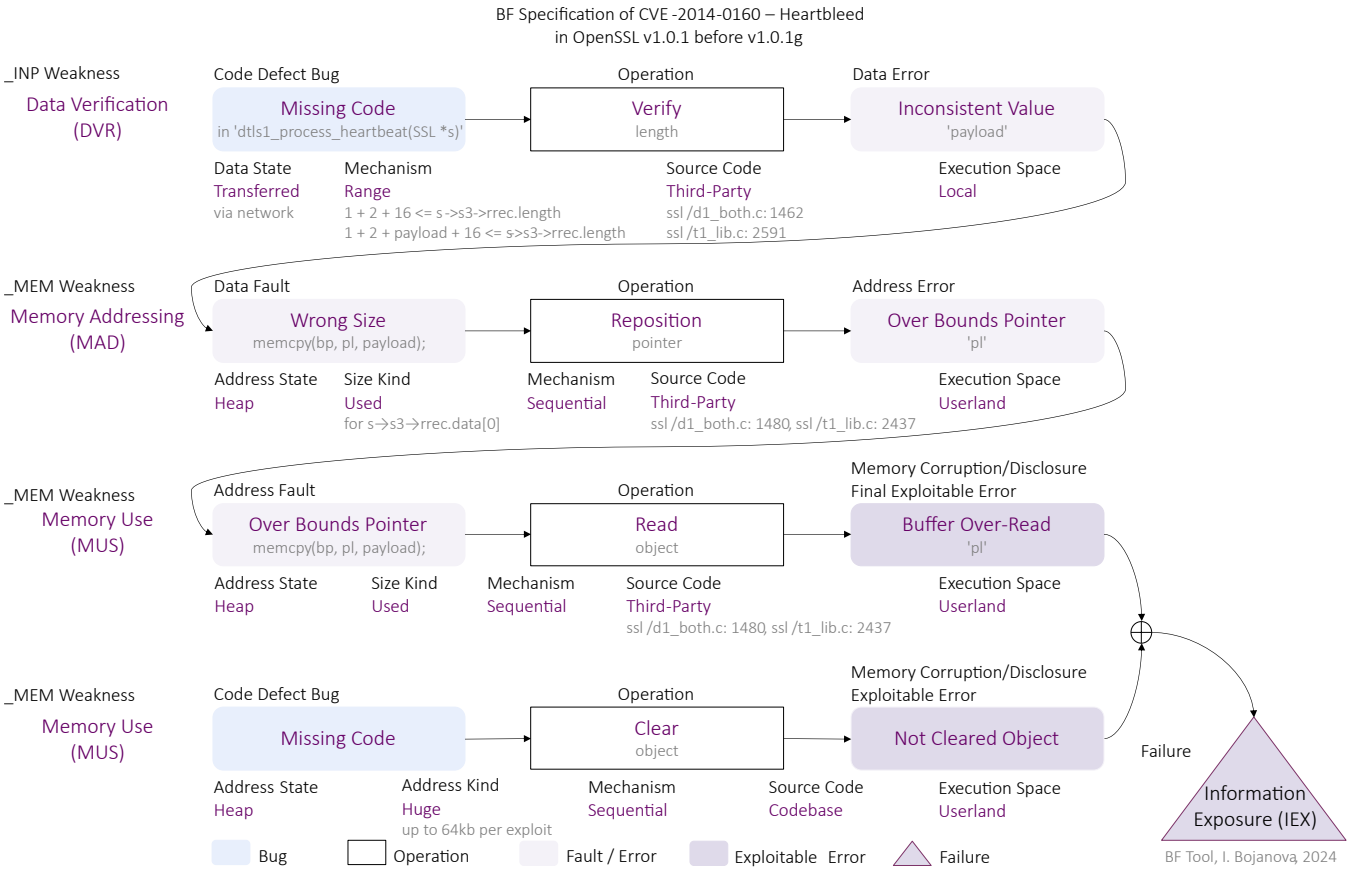

**Fig. 18.** BF GUI tool – utilizes the BF taxonomy and enforces the BF formal language syntax and semantics. Screenshots show the comprehensive BF labels for CVE-2014-0160 Heartbleed.

```
<?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.

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




| 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.

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 weakness 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*, *Operand*, *Error/Final 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 can enforce the initial weakness to start with a Bug, the rest of the weaknesses to start with a Fault – this is not necessary for partial specifications or if a vulnerability starts with a fault from a physical defect. 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 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 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.

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 800-231F BF Databases, Tools, and APIs.

## 9.3. BF Datasets

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 of security bugs and related faults enabling unambiguous specification of security weaknesses and vulnerabilities, BF offers a prominent new approach toward systematic creation of comprehensively labeled weaknesses and vulnerabilities datasets utilizing the BF taxonomy.

### 9.3.1. BFCWE

There are 938 CWE weakness types as of March 2024 [1]. NVD uses the 130 ”most commonly seen weaknesses” [34] 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) [18], 157 – to BF Input/Output Check (_INP) [17], and 60 – to BF Memory Corruption/Disclosure (_MEM) [16] 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. Most of them represent also the most dangerous weakness types – injection and memory corruption/disclosure [35].

The methodology utilizing BF is as follows:

1. CWE2BF Mappings: Basic computer science and security 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)*, *(bug, operation, error, operation, final error)*, or *(fault, operation, final error)* weakness triples (see [16–18, 20, 25]).
2. BF Specifications: The BFCWE tool generates BFCWE formal specifications as entries of the BFCWE 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 would augment NVD database and the CWE repository (to the extend possible) by adding formal BF specifications of possible BF weakness triples. Note that BF has the expressive power to describe any security weakness and not only the types listed in CWE.

For details on the BFCWE dataset refer NIST SP 800-231G BF Weakness and Vulnerability Datasets.

### 9.3.2. BFCVE

There are over 170 000 CVEs labeled with CWEs by NVD as of March 2024. 3 329 of them map by final weakness to the BF Data Type (_DAT) [18], 63 172 – to Input/Output (_INP) [17], and 40 454 – to Memory Corruption/Disclosure (_MEM) [16] class types. 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. Most of them represent also vulnerabilities related to the most dangerous weakness types – injection and memory corruption/disclosure [35].

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 error)* or *(bug, operation, final 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 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 security vulnerabilities.

The BFCVE dataset would augment the NVD database and the CVE repository supplying the formal BF specifications of CVE entries. However, BF has the expressive power to describe any security weakness and vulnerability and not only the listed in CWE and CVE. BF has its own databases with causal weakness taxonomies and formal vulnerability specifications built by their underlying weaknesses specifications.

For details on the BFCVE dataset refer NIST SP 800-231G BF Weakness and Vulnerability Datasets.

## 9.4. BF Vulnerability Classification

The Bugs Framework (BF) and the development of BFCWE and BFCVE datasets would allow the impossible in 1999 "more complex representations" of vulnerabilities in contrast to the "1-dimensional representation" provided by the CVE enumeration [12]. The BF Vulnerability Classification Model (see Fig. 22) details how the BF taxonomy and the BF tools can be utilized for generation of BFCWE and BFCVE datasets (see also Sec. 9.3.1 and Sec. 9.3.2) and by themselves or by querying other vulnerability repositories for generation of diverse multidimensional vulnerability classifications based on common properties and similarities.

A key part of the BFCVE dataset generation is the use of preliminary sets of partial BF specifications of CVEs for which "Code with Fix" is available. These CVE sets are generated by querying NVD and the corresponding GitHub repository towards BFDB. For example, Fig. 21 shows the SQL query for vulnerabilities related to the Memory Corruption/Disclosure BF class type towards the DiverseVul [33] fix commits extracted via the GitHub REST API.

```
with cweClass as (
select distinct c.Type, class = c.Name, wo.cwe
from bf.class c
inner join bf.operation o on c.Name = o.Class
inner join cwebf.operation wo on o.Name = wo.operation
)
select m.cve [CVE], m.cwe [CWE], n.score [CVSS], ci.url [CodeWithFix], c.Type [BFClassType],
       c.class [BFClass], v.cause [Cause], v.operation [Operation], v.consequence [Consequence]
from cweClass c
inner join nvd.mapCveCwe m on m.cwe = c.cwe
inner join nvd.cve n on m.cve = n.cve
inner join gitHubVul.cve u on u.cve = n.cve
inner join gitHubVul.commitId ci on ci.id = u.commitId
inner join cwe.cwe w on w.id = m.cwe
inner join cwebf.specification s on s.cwe = m.cwe
inner join cwebf.mainWeakness mw on mw.mainWeakness = s.mainWeakness
inner join bf.validWeakness v on v.id = mw.weakness
left outer join cwebf.otherWeakness cw on cw.cwe = m.cwe and cw.mainWeakness = s.mainWeakness
left outer join bf.validWeakness vv on vv.id = cw.weakness
left outer join bf.operation oo on oo.Name = vv.operation
left outer join bf.class cc on oo.Class = cc.Name
where (c.Type = '_MEM')
order by  n.score desc, m.cve, s.cwe, cw.chainId
```

**Fig. 21.** GitHub-NVD-BF SQL Query producing the set of CVEs related to the Memory Corruption/Disclosure BF class type and for which the "Code with Fix" is available.

the security experts and/or LLMs code analysis needed to create complete BF vulnerability specifications. Several security teams are conducting research in this direction.

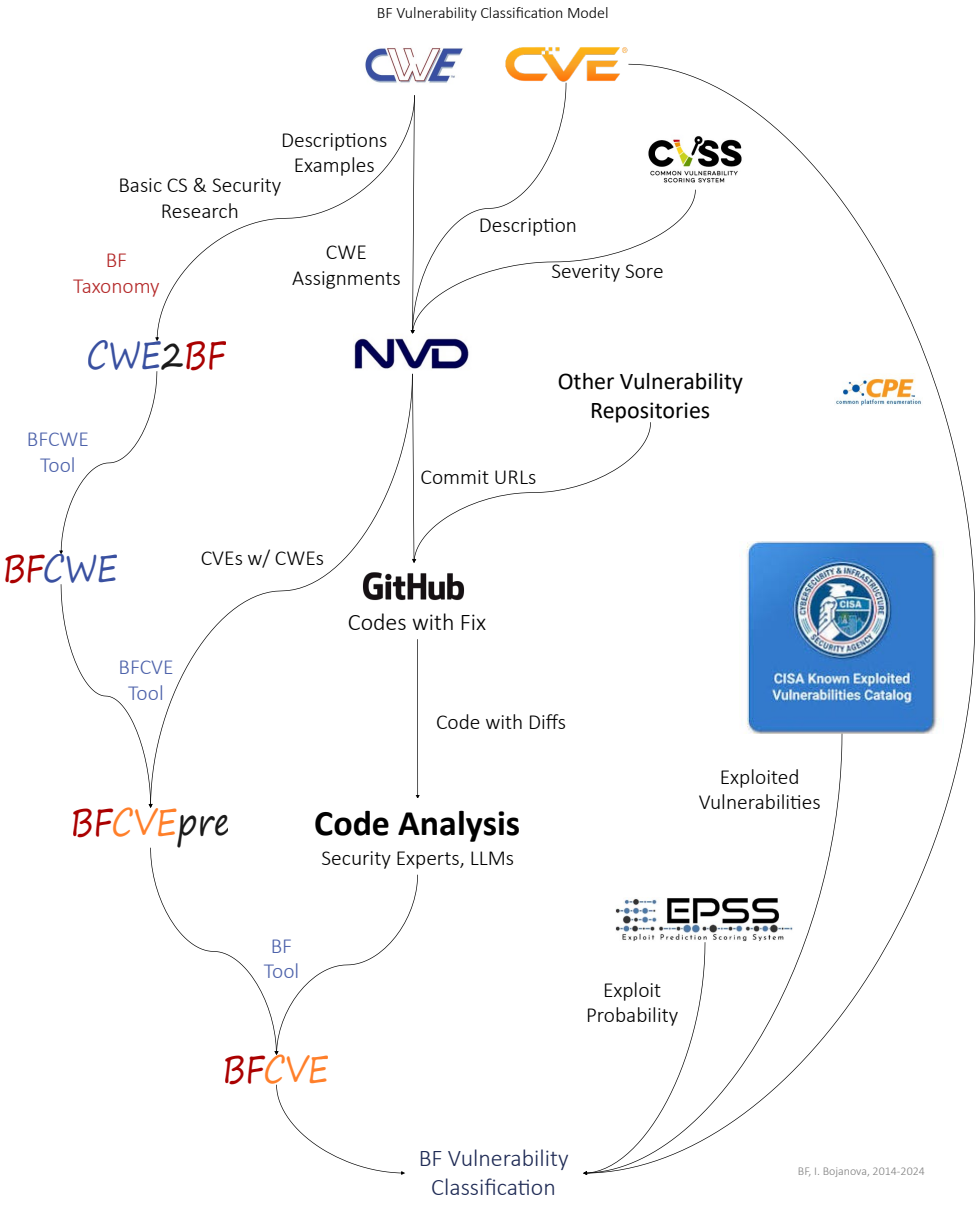



**Fig. 22.** BF Vulnerability Classification Model.

Security vulnerabilities could be classified by common root causes (software or firmware bugs, or hardware defect induced faults) – e.g., by wrong data type used in a variable declaration, by propagating faults, or by common final errors. 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. Their analysis would allow deeper research on most significant [35] and most exploited [36] weaknesses and vulnerabilities.

Vulnerabilities could be classified also by the number of their 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 (e.g. floats safety), input/output safety, and memory safety coding problems related to particular vendors and products. The BF Vulnerability Classification API will be providing access to the latest developments.

These multi-dimensional BF vulnerability classifications would contribute to the deeper analysis and refined understanding of security weaknesses, vulnerabilities, exploits, and failures. They would allow focused cybersecurity research and highly informed development of effective countermeasures against security potential threats and specific attacks.

## 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.

[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] 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.

[6] 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

[7] 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

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

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

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

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

[12] 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.

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

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

[15] 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

[16] Irena Bojanova, C E Galhardo (2021) Classifying Memory Bugs Using Bugs Framework Approach. *2021 IEEE 45nd Annual Computer, Software, and Applications Conference (COMPSAC)*, pp 1157–1164. https://doi.org/10.1109/COMPSAC51774.2021.00159

[17] 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

[18] 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

[19] 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.

[20] 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

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

[22] 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

[23] 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

[24] 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

[25] 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/10.1

109/MITP.2023.3259119
[26] Irena Bojanova (2020-2024) NIST Bugs Framework (BF), BF Tool. Available at https://usnistgov.github.io/BF/info/tools/bf-tool.
[27] Donald Knuth (1968) Semantics of context-free languages. *Math. Systems Theory 2*, p 127–145. https://doi.org/10.1007/BF01692511
[28] GitHub (2008) GitHub. Available at https://github.com.
[29] NIST (2005-2024) Software Assurance Reference Dataset (SARD). Available at https://samate.nist.gov/SARD.
[30] FIRST (2021-2024) Exploit Prediction Scoring System (EPSS). Available at https://www.first.org/epss.
[31] Irena Bojanova (2020-2024) NIST Bugs Framework (BF), BFCVE Tool. Available at https://usnistgov.github.io/BF/info/tools/bfcve-tool.
[32] Irena Bojanova (2020-2024) NIST Bugs Framework (BF), BFCWE Tool. Available at https://usnistgov.github.io/BF/info/tools/bfcwe-tool.
[33] 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.
[34] MITRE (2023) Weaknesses for Simplified Mapping of Published Vulnerabilities. Available at https://cwe.mitre.org/data/definitions/1003.html.
[35] Carlos EC Galhardo, P Mell, I Bojanova, A Gueye (2020) Measurements of the Most Significant Software Security Weaknesses. *2020 Annual Computer Security Applications Conference (ACSAC)*, p 154–164. https://doi.org/10.1145/3427228.3427257
[36] Peter Mell, I Bojanova, Carlos EC Galhardo (2024) Measuring the Exploitation of Weaknesses in the Wild. *xxx*, p xx–xx. https://doi.org/xxx