EDITOR: Irena Bojanova, irena.bojanova@computer.org

DEPARTMENT: CYBERSECURITY

Comprehensively Labeled Weakness and Vulnerability Datasets via Unambiguous Formal Bugs Framework (BF) Specifications

Irena Bojanova, https://orcid.org/0000-0002-3198-7026, *NIST, Gaithersburg, MD, 20899, USA*

Abstract—The current state of the art in software security – describing weaknesses as Common Weakness Enumeration (CWE) entries and vulnerabilities as Common Vulnerabilities and Exposures (CVE) entries, and labeling CVEs with CWEs—is not keeping up with the modern cybersecurity research and application requirements for comprehensively labeled datasets. As a formal classification system of software security bugs and related software faults enabling unambiguous specification of software security weaknesses and vulnerabilities, the NIST Bugs Framework (BF) offers a prominent new approach toward systematic creation of weakness and venerability datasets labeled with the BF taxonomy. This work presents methodologies based on the BF formal language and developed BF tools for comprehensive labeling of common weaknesses – including CWEs – and publicly disclosed vulnerabilities – including CVEs. The developed taxonomic datasets, transformation algorithms, databases, and queries can support a new range of research and implementation efforts for weakness and vulnerability specification generation, bug detection, vulnerability identification and remediation/mitigation, and test-case generation.

Cybersecurity of critical infrastructure and software supply chains is an increasingly pressing societal challenge. Attacks on cyberspace are not only growing, they are also more sophisticated and more dangerous. Modern cybersecurity research and application must overcome them and assure software security vulnerability prevention, remediation, or mitigation. However, the current state of art for specifying and labeling software security weaknesses and vulnerabilities is not keeping up with their requirements. There is a critical need for formal approaches in classifying software security bugs and systematic comprehensive labeling of common weaknesses and disclosed vulnerabilities. The NIST Bugs Framework (BF) [1] is a formal classification system of software security bugs and related software faults enabling unambiguous formal specification of software security weakness types and vulnerabilities, which aims to address these necessities.

Widely used sources of weakness and vulnerability descriptions are the Common Weakness Enumeration (CWE) [2] and the Common Vulnerabilities and Exposures (CVE) [3] repositories. The National Vulnerabilities Database (NVD) [4] also labels the CVEs with CWEs. However, the CWE and CVE descriptions are in natural language and, thus, far from formal. The CWE to CVE assignments could be challenging and ambiguous, as CWEs may be overly specific, unclear, or overlapping [5].

As of January 2024, there are 934 CWE weakness

Digital Object Identifier 10.1109/XXX.0000.0000000
Date of current version xx July 2023.

Disclaimer: Certain equipment, instruments, software, or materials, commercial or non-commercial, are identified in this paper in order to specify the experimental procedure adequately. Such identification is not intended to imply recommendation or endorsement of any product or service by NIST, nor is it intended to imply that the materials or equipment identified are necessarily the best available for the purpose.

types and 235 625 CVEs, including 169 819 CVEs labeled with CWEs by NVD. Thirty percent of the CWEs and most of the CVEs (63%) map to the BF Data Type (_DAT) [6], BF Input/Output Check (_INP) [7], and BF Memory Corruption/Disclosure (_MEM) [8] bugs class types, providing the base for comprehensively labeled BFCWE and BFCVE datasets.

This work presents BF-based methodologies and tools (see Tools at [1]) for systematic comprehensive labeling of common weakness types and disclosed vulnerabilities. The BFCWE tool facilitates the creation of CWE-to-BF (CWE2BF) mappings by weakness operation, error, and final exploitable error, and possibly by entire main `(cause, operation, consequence)` BF weakness triple and generates BFCWE formal specifications as entries for the BFCWE dataset and graphical representations of the mappings and the specifications. The BFCVE tool generates possible chains of weaknesses for a vulnerability by identified failure and final exploitable error or possibly entire final exploitable weakness, generates possible BFCVE formal specifications and their graphical representations, and identifies and recommends a CWE(s) for NVD assignment. Code analysis and the BF GUI (Graphical User Interface) tool are used to identify and complete the unique unambiguous BF vulnerability specifications.

The tools utilize machine-readable representations of the current BF taxonomy in XML or JSON formats, generated from the BF relational database. The tools also utilize the created BFVul mashup database with weakness and vulnerability data from the CWE, CVE, NVD, and GitHub [9] vulnerability repositories. Further vulnerability analysis can be performed utilizing the extension of BFVul with data from Known Exploited Vulnerabilities Catalog (KEV) [10], Exploit Prediction Scoring System (EPSS) [11], Software Assurance Reference Dataset (SARD) [12],

The developed taxonomic datasets and transformation algorithms, and databases and queries can support a new range of research and implementation efforts for weakness and vulnerability specification generation, bug detection (triaging), vulnerability remediation or mitigation, and test-case generation.

Current State of the Art

The current state of the art in describing and mapping software security weaknesses and vulnerabilities involves the use of the Common Weakness Enumeration (CWE) [2], the Common Vulnerabilities and Exposures (CVE) [3], and the National Vulnerabilities Database (NVD) [4]. CWE is a community-developed list of software and hardware weakness types with descriptions, examples, and references – 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 cybersecurity vulnerabilities with descriptions and references – each CVE entry is assigned a $CVE-YYYY-X$ ID, where $YYYY$ is the year of disclosure and X is of four or five digits. NVD maps CVEs to CWEs while also assigning Common Vulnerability Scoring System (CVSS) [13] severity scores.

Although widely used, CWE, CVE, and NVD face certain issues. Many CWEs and CVEs have imprecise descriptions and unclear causality, and CWE exhibits gaps and overlaps in coverage. Additionally, there are no strict methodologies for chaining the weaknesses underlying a vulnerability and for backwards bug identification from a failure, and there are no tools to assist users in creating and graphically visualizing weakness and vulnerability descriptions.

The NIST Bugs Framework (BF) offers a new prominent approach for addressing these challenges. It possesses the expressive power to formally describe any software bug or weakness underlying any vulnerability. Thus, it could be utilized to augment CWE, CVE, and NVD with unambiguous weaknesses and vulnerabilities BF specifications.

The Bugs Framework (BF)

The Bugs Framework (BF) [1] is a formal classification system of software security bugs and related software faults enabling unambiguous specification of software security weaknesses and underlined by them vulnerabilities. It comprises software security concepts definitions, Bugs Models with possible flow of operations, a structured orthogonal (not overlapping) by operation taxonomy, a Vulnerability State Model – as a chain of a bug and faults states leading to a failure(s), and a Vulnerability Specification Model – as a chain of weakness triples adhering to causation and propagation rules. BF addresses the CWE and CVE challenges via its classification, causation, propagation (chaining), and failure-to-bug identification features. The BF taxonomy, and causation and propagation rules form the BF LL(1) (Left-to-right Leftmost-derivation One-symbol-lookahead) Formal Language, which forms the backbone of the BF Tools.

In contrast to CWE's exhaustive enumeration approach, BF adopts a structured classification. Each BF class is a taxonomic category defined by a set of operations, valid `cause-operation→consequence` within-weakness transitions, and attributes. It relates to a specific software execution phase, the possible

operations' bugs and operands' faults defining the causes, and consequences resulting from the operations over the operands. BF describes a weakness as an instance of a taxonomic BF class with one *operation*, one *cause*, one *consequence*, and their *attributes*. BF describes a vulnerability as a chain of BF class instances (weaknesses) and their `consequence→cause` between-weaknesses transitions. A software security *bug* causes the first weakness, leading to an *error*. This error becomes the *fault* causing another weakness and propagates through subsequent *errors* until a *final exploitable error* is reached, causing a security *failure*. Causation and propagation between weaknesses are by valid operation flow and `error→fault` transitions. Causation and propagation between vulnerabilities are by valid `exploit result→fault` transitions. The BF specification of a vulnerability adheres to the BF causation and propagation rules to form a chain of underlying weaknesses leading to a failure. This allows nuanced understanding of the vulnerability and unambiguous failure to bug identification – going backwards through all the `consequence→cause` transitions at execution, i.e., by improper (faulty) operand until an operation is improper (has a bug). Fixing the bug within that operation remediates the vulnerability.

BF Vulnerability Specification Model

The Bugs Framework (BF) models a vulnerability specification (see Figure 1) 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 and the BF vulnerability state model at Bojanova [1].

Causation within weaknesses is by meaningful `(cause, operation, consequence)` weakness triples defined for each BF class 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)`, `(bug, operation, final exploitable error)`, `(fault, operation, error)`, or `(fault, operation, final exploitable error)` weakness triple. Causation between weaknesses is by valid operation flow and valid `consequence→cause` transitions adhering to the BF Bugs Models and the appropriate BF LL(1) semantics. More specifically, as graphs of meaningful $(operation_1, \ldots, operation_n)$ bug or fault state paths and matrices of meaningful `consequence→cause` transitions. Causation between vulnerabilities is by an `exploit→fault-operation` transitions – the result from an exploit becomes the fault for a new faults-only vulnerability.

Propagation between weaknesses is by valid `error→fault` transitions – the error resulting from the operation of a weakness becomes the fault of another weakness. The matching is by fault value or only by fault type and valid `consequence→cause` transitions – typical for for weaknesses of BF class types on different levels of abstraction. Propagation between vulnerabilities is by `exploit result` matching the `fault` type of the chained vulnerability.

For simplicity, Figure 1 does not show vulnerability convergence and chaining, as they directly reflect the corresponding transitions from the BF Vulnerability States Model at [1].

BFCWE Dataset

There are 934 CWE weakness types as of January 2024 [2]. NVD uses the 130 "most commonly seen weaknesses" [14] for labeling CVEs, but it might also list other CWEs as assigned by third-party contributors.

Of the 934 CWEs, 72 map to BF Data Type (_DAT) [6], 157 – to BF Input/Output Check (_INP) [7], and 60 – to BF Memory Corruption/Disclosure (_MEM) [8] bugs class types. These 289 unique CWEs form 30% of the CWE repository and provide the base for systematic creation of a comprehensively labeled software weaknesses BFCWE dataset via utilizing BF.

The methodology is as follows: (1) Basic software security weaknesses research and meticulous analysis of the natural language descriptions of all data type, input/output check, and memory related CWEs (as well as of relevant code examples and CVEs) is conducted to create CWE2BF mappings by weakness operation, error, final error and then by detailed BF `(bug, operation, error)`, `(fault, operation, error)`, or `(fault, operation, exploitable error)` weakness triples (see Bojanova and Guerrerio for _MEM [15]). (2) The BFCWE tool generates BFCWE formal specifications as entries of the BFCWE software security weakness types dataset. (3) 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. This same methodology would also reveal the overlaps in CWEs.

For example, the possible main-weakness triples for CWE-125 are `(Over Bounds Pointer,`

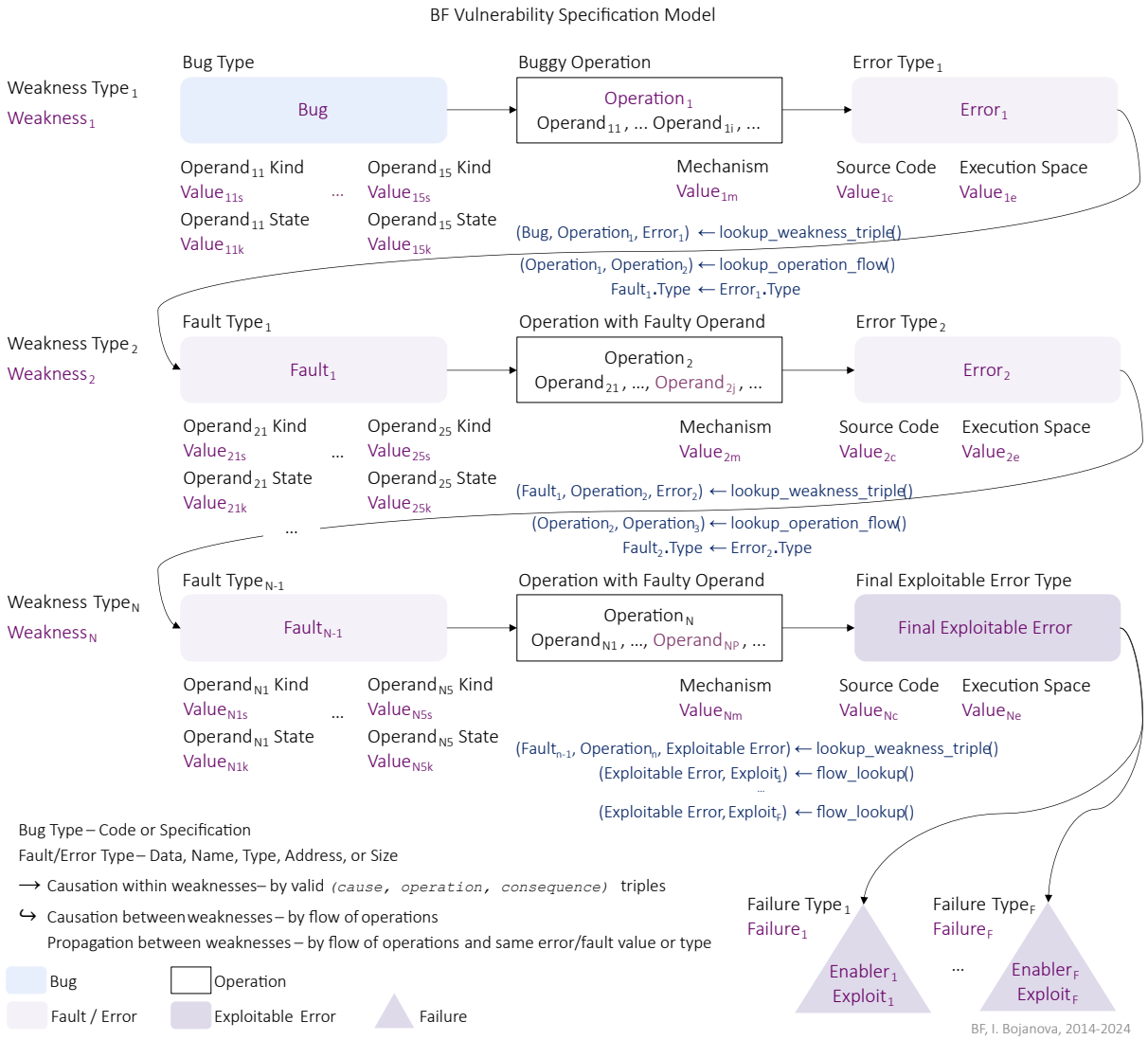

FIGURE 1: BF Vulnerability Specification Model. Reflects the BF Taxonomy `(cause, operation, consequence)` structure, and the BF Formal Language syntax and semantics. Possible types of triples are `(bug, operation, error)`, `(bug, operation, final exploitable error)`, `(fault, operation, error)`, or `(fault, operation, final exploitable error)`.

`Read, Buffer Over-Read)` and `(Under Bounds Pointer, Read, Buffer Over-Read)`. Figure 2 illustrates the 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 [15] 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. All developed BFCWE specifications are added to the comprehensively labeled BFCWE dataset at Bojanova [1].

This same methodology helps reveal CWEs overlaps, as many CWEs have the same BF specifications. Although, a CWE should be about a single weakness, the descriptions of some CWEs also reveal possible causing chains of weakness triples (see Bojanova and Guerrerio for _MEM [15]). 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 software security vulnerabilities.

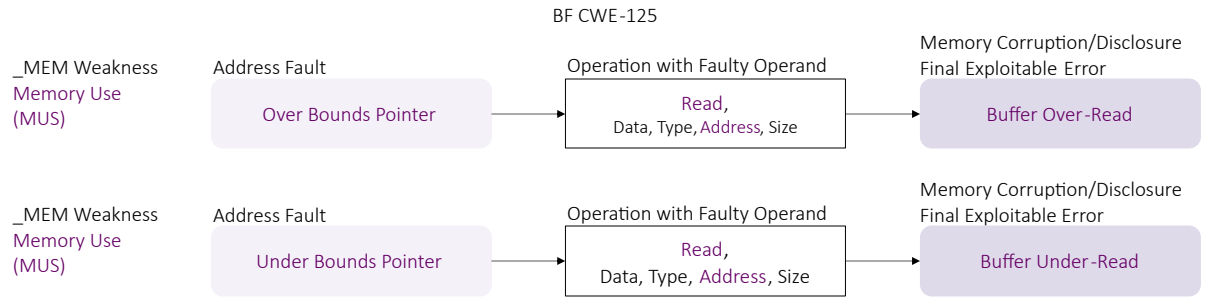

FIGURE 2: BF Memory Use (MUS) weakness specifications of CWE-125.

The BFCWE dataset could augment the CWE repository and the NVD database by adding the formal BF specifications of possible BF weakness triples for each CWE entry.

BFCVE Dataset

There are 169 819 CVEs labeled with CWEs by NVD as of January 2024. Most of them map by final exploitable weakness to the BF Data Type (_DAT) [6], Input/Output (_INP) [7], and Memory Corruption/Disclosure (_MEM) [8] bug/weakness types: 3 329 map to _DAT, 63 172 – to _INP, and 40 454 – to _MEM. These 106 955 unique CVEs represent 63% of the CVEs labeled with CWEs by NVD, providing the base for systematic creation of a comprehensively labeled BFCVE software security vulnerability dataset via utilizing BF.

The methodology is as follows: (1) The BFCVE tool identifies CVEs with assigned CWEs for which *Code with Fix* is available. (2) The BFCVE tool generates possible chains of weaknesses for a vulnerability – by identified failure and some or all the elements of the final weakness – `(fault, operation, final exploitable error)` or (bug, operation, final exploitable error) in the case of one weakness vulnerability – utilizing the BF taxonomy, and syntax and semantic rules. (3) Code analysis and the BF GUI tool are used to filter the generated chains and complete the unambiguous BF vulnerability specifications. (4) The BFCVE tool generates graphical representations for enhanced understanding of the BF vulnerability specifications as entries for the BFCVE software security vulnerability dataset. (5) The BFCVE tool identifies, refines, and recommends a CWE(s) for NVD assignment.

On step (1), 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 (see the SQL query in Figure 3). For example, there are 269 CVEs in DiverseVul [16] for which final weaknesses map to BF Memory Corruption/Disclosure weakness triples, and the *Code with fix* can be extracted from the fix commits via the GitHub REST API.

On step (2), information on the failure(s) and the final exploitable weakness is gained from the CVE report(s), the CVE description, and the CWE2BF weakness triple mappings if a CWE(s) is assigned by NVD. The BFCVE tool utilizes the BF relational database and the NVD REST API to extract the CWE2BF triples for that CVE. Then, the BFCVE tool applies the BF causation and propagation rules (i.e., the BF formal language syntax and semantics) to go backwards from the failure(s) through the final exploitable weakness to generate all possible BF chains of weaknesses for that specific CVE, independently of whether the CVE *Code with Fix* is available.

Going backwards from the failure, the BFCVE tool builds a connected acyclic undirected graph (a tree, which root is the failure) of all possible weakness chains with type-based `fault-to-error` backward propagation, plus for weaknesses of same BF class type – with name-based backward propagation. Then the chains undergo scrutiny to ensure further alignment with the BF Formal Language semantics – the Causation Matrix of all meaningful `(cause, operation, consequence)` weakness triples and the Propagation Graphs of meaningful $(operation_1, \ldots, operation_n)$ bug or fault state paths and Matrix of all valid `consequence→cause` transitions between weaknesses.

Identified beforehand failure(s) and final exploitable weakness triple(s) reduce dramatically the number of generated possible paths in the acyclic graph. Step (2) is also a good starting point for specifying vulnerabilities not recorded in CVE, as far as failure(s) and final exploitable weakness information are identifiable. Identified beforehand failure(s) and final exploitable weakness triple(s) reduce dramatically the number of generated possible paths in the acyclic graph. Step

(2) is also a good starting point for specifying vulnerabilities not recorded in CVE, as far as failure(s) and some of the final exploitable weakness information are identifiable.

On step (3), the CVE *Code with Fix* is examined towards the generated chains of weakness triples to pinpoint the unique unambiguous BF vulnerability specification. For that both the BF tool functionality and static/dynamic analysis or Large Language Models (LLMs) can be utilized.

```
with cwe2bf 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 cwe2bf 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
```

FIGURE 3: 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.

For example, CVE-2014-0160 Heartbleed – a vulnerability in the OpenSSL cryptographic software library – is mapped in NVD to CWE-125 [17]. The CWE2BF mappings for CWE-125 restricts to two the final exploitable weakness options for Heartbleed: `(Over Bounds Pointer, Read, Buffer Over-Read)` or `(Under Bounds Pointer, Read, Buffer Under-Read)`. However, the CVE-2014-0160 description reveals the word *over*, which points CWE-125 is too abstract for it and eliminates the second BF final exploitable error option. In addition, as Heartbleed leads to information exposure, the last part of the BF weaknesses chain specification is: `(Over Bounds Pointer, Read, Buffer Over-Read)→Information Exposure (IEX).` Note that the `Read` operation uniquely identifies the BF MUS class, as BF classes do not overlap by operation [8].

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 Figure 4, the failure is the root, the final exploitable error is the first node, and a bug is the last node in each path. The only options for the weakness causing the final exploitable weakness are: `(Wrong Index, Reposition, Over Bounds Pointer)` and `(Wrong Size, Reposition, Over Bounds Pointer).` Both of them have the same options for causing chains, only two of which do not start with a bug, but even for them the preceding weakness options start with a bug. Exhausting these few options via source code analysis or use of LLMs, it is 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).`

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).`

For step (4), the BFCVE tool generates graphical representation(s) for enhanced understanding of the BFCVE formal specifications.

Although step (5) seems illogical, as a BF specification already provides comprehensive information, it may be useful, for example, when comparing CWE-based testing tool reports or if a more appropriate CWE is identified.

Developed BFCVE specifications are added to the comprehensively labeled BFCVE dataset at Bojanova [1]. The BF semantic graphs and matrices and the datasets are also continuously enriched from newly developed formal BF specifications of CVEs and other reported software security vulnerabilities.

The BFCVE dataset could augment the CVE repository and the NVD database by supplying the formal BF specifications of CVE entries.

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

FIGURE 4: BFCVE tool generated tree of possible chains for CVE-2014-0160 (Heartbleed) using the BF methodology for backwards Bug identification from a Failure.

BF GUI Tool

Currently, the BF tool is a desktop application (see Figure 5), 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 Figure 6), and open and browse previously created *.bfcve* specifications.

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

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

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

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

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

Conclusion

The presented systematic approach for creating comprehensively labeled BFCWE and BFCVE datasets, involving the BFCWE, BFCVE, and the BF API and GUI tools contributes to a refined understanding and deeper analysis of software security weaknesses and

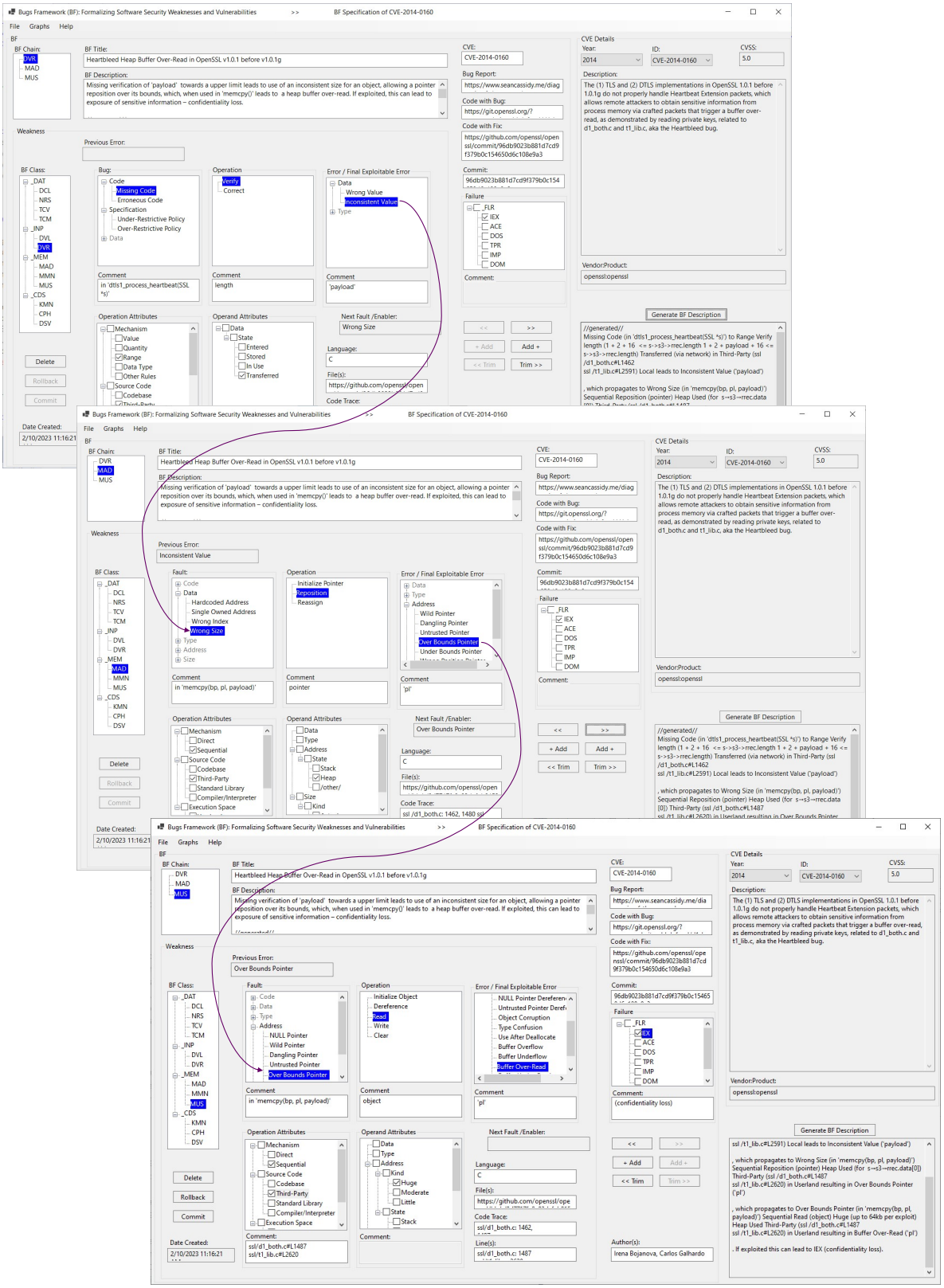

FIGURE 5: BF GUI tool – utilizes the BF taxonomy and enforces the BF LL(1) formal language syntax and semantics. Screenshots show the comprehensive BF labels for CVE-2014-0160 Heartbleed.

```
<?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 verification of 'pay
  <DefectWeakness Class="DVR" ClassType="_INP" Language="C" File="https://github.com/openssl/openssl/commit/96db9023b881d7cd9f379b0c154650d6c
    <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  <= s->s3->rrec.length 1 + 2 + payload + 16 <= s->s3->rrec.length" Type="Mechanism">Range</Attribute>
        <Attribute Comment="ssl/d1_both.c#L1462 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/0d7717fc9c83dafab8153cbd5e2180e6e04cc802/
    <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 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/0d7717fc9c83dafab8153cbd5e2180e6e04cc802/
    <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 ssl/t1_lib.c#L2620" Type="Source Code">Third-Party</Attribute>
        <Attribute Type="Execution Space">Userland</Attribute>
      </Operation>
    </Attributes>
  </Weakness>
  <Failures ClassType="_FLR">
    <Cause Type="Memory Corruption/Disclosure">Buffer Over-Read</Cause>
    <Failure Class="IEX" Comment="(confidentiality loss)" />
  </Failures>
</BFCVE>
```

FIGURE 6: BFCVE specification of CVE-2014-0160 Heartbleed in XML format.

vulnerabilities and would aid in the development of effective security countermeasures. The BFCWE tool facilitates generation of unambiguous formal BF weakness specifications as entries of a comprehensively labeled BFCWE dataset. The BFCVE tool generates pre-labeled vulnerability datasets to be further refined via code analysis. The BF GUI tool guides the creation of unambiguous formal BF specifications as entries of a comprehensively labeled BFCVE dataset.

The BFCWE dataset could augment the CWE repository and the NVD database by supplying formal BF specifications of possible BF weakness triples for each CWE entry. The BFCVE dataset could augment the CVE repository and the NVD database by supplying unambiguous comprehensive formal BF specifications of CVE entries. The developed taxonomic datasets, transformation algorithms, databases, and queries would benefit the implementation of a new range of bug detection (triaging), test-case generation, and weakness and vulnerability specification generation tools. A BFVul API[1] will be made available for retrieving information from the BFVul mashup database

with weakness and vulnerability data from the CWE, CVE, NVD, KEV [10], EPSS [11], SARD [12], and GitHub [9] vulnerability repositories.

Please feel free to join the BFCWE, BFCVE, and BFAI Challenges at [1] and to contact BF's Principal Investigator (PI), Irena Bojanova, to discuss ideas for collaboration.

Irena Bojanova, is a computer scientist at NIST, Geitherburg, MD 20899, USA. Contact her at irena.bojanova@nist.gov.

References

[1] Irena Bojanova, NIST, *Bugs Framework (BF)*, 2014-2024. [Online]. Available: https://www.nist.gov/itl/ssd/software-quality-group/samate/bugs-framework.

[2] MITRE, *Common Weakness Enumeration (CWE)*, 2006-2024. [Online]. Available: https://cwe.mitre.org.

[3] MITRE, *Common Vulnerabilities and Exposures (CVE)*, 1999-2024. [Online]. Available: https://cve.mitre.org.

[4] NIST, *National Vulnerability Database (NVD)*, 1999-2024. [Online]. Available: https://nvd.nist.gov.

[5] I. Bojanova and C. E. Galhardo, "Bug, Fault, Error, or Weakness: Demystifying Software Security Vulnerabilities," *IEEE IT Professional*, vol. 25, no. 1, pp. 7–12, 2023. DOI: 10.1109/MITP.2023.3238631.

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

[7] I. Bojanova and C. E. Galhardo, "Input/Output Check Bugs Taxonomy: Injection Errors in Spotlight," in *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2021, pp. 111–120. DOI: 10.1109/ISSREW53611.2021.00052.

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

[9] GitHub, *GitHub*, 2008-2024. [Online]. Available: https://github.com/.

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

[11] FIRST, *Exploit Prediction Scoring System (EPSS)*, 2021-2024. [Online]. Available: https://www.first.org/epss/.

[12] NIST, *Software Assurance Reference Dataset (SARD)*, 2024. [Online]. Available: https://samate.nist.gov/SARD/.

[13] FIRST, *Common Vulnerability Scoring System*, 2015-2024. [Online]. Available: https://www.first.org/cvss.

[14] MITRE, *Weaknesses for Simplified Mapping of Published Vulnerabilities*, 2023. [Online]. Available: https://cwe.mitre.org/data/definitions/1003.html.

[15] I. Bojanova and J. J. Guerrerio, "Labeling Software Security Vulnerabilities," *IEEE IT Professional*, vol. 25, no. 5, pp. 64–70, 2023. DOI: 10.1109/MITP.2023.3314368.

[16] Y. Chen and at all, *DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection*, 2023. [Online]. Available: https://github.com/wagner-group/diversevul.

[17] I. Bojanova and C. E. Galhardo, "Heartbleed Revisited: Is it just a Buffer Over-Read?" *IEEE IT Professional*, vol. 25, no. 2, pp. 83–89, 2023. DOI: 10.1109/MITP.2023.3259119.