NISTIR 8304

SATE VI Ockham Sound Analysis Criteria

Paul E. Black
Kanwardeep Singh Walia

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8304

National Institute of
Standards and Technology
U.S. Department of Commerce

NISTIR 8304

SATE VI Ockham Sound Analysis Criteria

Paul E. Black
Software and Systems Division
Information Technology Laboratory

Kanwardeep Singh Walia
California State University
Sacramento, California

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8304

May 2020

U.S. Department of Commerce
Wilbur L. Ross, Jr., Secretary

National Institute of Standards and Technology
Walter Copan, NIST Director and Undersecretary of Commerce for Standards and Technology

National Institute of Standards and Technology Interagency or Internal Report 8304
Natl. Inst. Stand. Technol. Interag. Intern. Rep. 8304, 46 pages (May 2020)

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8304

Abstract

Static analyzers examine the source or executable code of programs to find problems. Many static analyzers use heuristics or approximations to examine programs with millions of lines of code for hundreds of classes of problems. The Ockham Sound Analysis Criteria recognizes static analyzers that are precise. In brief the criteria are (1) the analyzer's findings are claimed to always be correct, (2) it produces findings for most of a program, and (3) even one incorrect finding disqualifies an analyzer. This document begins by explaining the background and requirements of the Ockham Criteria and how we determine if a tool satisfies it.

As part of Static Analysis Tool Exposition (SATE) VI, we examined two tools: Astrée and Frama-C with the Evolved Value Analysis (Eva) plug-in. Examining the tool outputs led us to find several systematic mistakes in the Juliet 1.3 test suite and thousands of mistakes in its manifest of known errors.

Our conclusion is that Astrée and Frama-C with Eva satisfied the SATE VI Ockham Sound Analysis Criteria.

Key words

Ockham Criteria; software assurance; software measurement; software testing; sound analysis; static analysis.

Table of Contents

Glossary

Term	Definition
bad function	A function in a Juliet test case that is written to exhibit a weakness. (Sec. 1.2.1)
good function	A function in a Juliet test case that is identical to a bad function, except that it does not have the weakness. (Sec. 1.2.1)
finding	A definitive statement by a tool about a specific place in code, e.g., the presence or absence of a weakness. (Sec. 2)
site	A location in code where a weakness might occur. (Sec. 1.2.2)
buggy site	A site that has a bug or weakness.
sound	Every finding is correct. (Sec. 2.1)
weakness	The property of a piece of code such that execution could lead to a fault. (Sec. 1.2.2)

Table 1. Glossary of Terms

1. Background

The Static Analysis Tool Exposition (SATE) is a recurring event at the National Institute of Standards and Technology (NIST) led by the Software Assurance Metrics And Tool Evaluation (SAMATE) team [1]. SATE aims to improve research and development of source code static analyzers, especially security-relevant aspects. To begin each SATE, the SAMATE team and other organizers select or create programs as test cases. Either participating tool developers or we run tools on the test cases and submit the results (tool reports). We then analyze the reports. Results and experiences are reported at the SATE workshop, and the final analysis is made publicly available.

The goals of SATE are to:

- Enable empirical research based on large test sets,
- Encourage improvement of tools, and
- Speed adoption of tools by objectively demonstrating their use on real software.

The general SATE procedure is:

1. We select test cases,
2. Tool makers run their tools and send the findings to us,
3. We analyze the tool reports, and
4. Parties share experience at a workshop.

See Ref. [2, Sec. 2.2] for details.

In SATE V [3], the SAMATE team introduced the Ockham Sound Analysis Criteria, a track for static analyzers whose analysis is logically sound. Tools that are not "bug-finders" can satisfy the Ockham Criteria, too. A tool that reports that pieces of code are certainly bug free is welcome.

We check that tools satisfy the SATE VI Ockham Sound Analysis Criteria to show that they are reliable and worth the effort to use. Beyond that, our test material and approach should help others investigate what assurance a tool provides for their own code in their own development process.

The rest of this section gives additional background and explains changes between the previous Ockham Sound Analysis Criteria evaluation and the current one. Section 2 explains the Criteria in detail. It also presents the general procedure we used to evaluate a tool by the Criteria. Section 3 explains details of the evaluation for Astrée, Sec. 3.1, and Frama-C, Sec. 3.2. Section 4 lists what we found and our conclusions.

1.1 Using Sound Static Analyzers

Our evaluation of tools against the Ockham Sound Analysis Criteria only reflects one aspect of using a sound static analyzer in a production software development process. Adding almost any tool to a software development process takes work. Even to evaluate as we have, there is a particular learning curve to effectively use sound and precise static analysis tools.

In order to be precise, such tools use a detailed description of the actual compilation and execution environments of the software being analyzed. Is an `int` 32 or 64 bits on the

target computer? Does the code rely on the compiler laying out memory for a `struct` in a certain order with no padding? Do you want warnings of unsigned short integer overflow if your code does a lot of masking and shifting, e.g., for hashes or encryption? Is the high-order bit propagated when a signed integer shifted right? How is floating-point addition rounded? The C11 standard [4] allows different definitions of `main()` and different behaviors of bitwise operators. The term "implementation-defined" occurs almost 200 times in the C11 standard.

In addition, the tools are elaborate systems with many abilities. As an analogy, consider that the word "vehicle" includes bicycles, dump trucks, and buses. All have wheels, can be steered, and transport something, but their design and uses are very different. Similarly, we evaluate only a small part of the tools' capabilities for the SATE VI Ockham Criteria. Astrée [5] has sophisticated graphical user interfaces to select hundreds of options, including checking Motor Industry Software Reliability Association (MISRA) guidelines, controling software checking, and displaying violations found in context. Frama-C [6] is an open source suite of tools to analyze software written in C, such as code slicing, dependency analysis, and enabling proofs that code satisfies functional specifications. Another tool, not in SATE VI Ockham, Kestrel Technology's CodeHawk-C, exposes the validity requirements—proof obligations (PO)—of every code fragment reporting that each PO is satisfied, violated, or cannot be proved.

Consider that Thales defines many levels of using formal verification for software assurance [7]. These levels are Stone—adhering to the SPARK [8] programming language—then Bronze—proving variable initialization and clear data flow—then Silver, Gold, and finally Platinum—proving that software meets its fully- and formally-specified requirements. Similarly, a knowledgeable user will "tune" the use of sophisticated tools, for instance, specify depth of recursion, number of loops to unroll, and analysis options so the tool produces the most useful result. It took us three or four full days of experimenting, reading, and guidance from tool makers to get tools reporting the errors that we were interested in. Even then we do not claim that our choices were optimal for a software production environment.

1.2 Differences Between SATE V and SATE VI Ockham Exercises

In this subsection, we examine differences between the SATE V Ockham evaluation procedure and that of SATE VI Ockham. For details of the SATE V evaluation, see Ref. [9].

We only evaluated Frama-C in the SATE V Ockham Sound Analysis Criteria. For this SATE VI, we evaluated two tools: Astrée [5] and a new version of Frama-C [6] with its Evolved Value Analysis (Eva) plug-in.

1.2.1 Known Bugs

SATE V Ockham used Juliet Version 1.2 test cases. Juliet cases are small, synthetic programs with deliberate bugs. Each case has a "bad" function exhibiting the bug and one or more "good" functions that have the same code as the bad function, but with

the bug fixed. Juliet was originally developed by National Security Agency's Center for Assured Software. They are available from the Software Assurance Reference Dataset (SARD) [10]. Evaluating the tool warnings we got during SATE V Ockham uncovered previously-unknown systematic errors in Juliet 1.2. We used that information and other problem reports to fix some two dozen systematic problems in Juliet code. We also added test cases for integer overflow and underflow using unary increment and decrement operators. These fixes and additions resulted in Juliet Version 1.3. For details of changes from Juliet 1.2 to 1.3 and some of the issues remaining in 1.3, see Ref. [11]. We and potential participants chose the Juliet 1.3 C test suite for this Ockham.

To evaluate warnings, we needed to know the classes and location of bugs designed into Juliet 1.2. We did not have an independent list. Instead, we used the source code, which has hints in comments. The hints are inconsistent. For example, the comment may be two lines or three lines before the bad code and sometimes both the bad and the good code have exactly the same comment. It took a lot of work to develop a program to locate comments, then produce consistent classes and locations for bugs. Frequent crosschecking against warnings drew our attention to some nuance or peculiarity in comment phrasing or location for certain test case variants. We embodied our final bug list in a *manifest* for the Juliet 1.3 C test suite.

1.2.2 Determining Sites

To explain sites, we first define "weakness". A piece of code has a *weakness* when some execution could lead to a fault. In contrast, a *vulnerability* in a system could be accidentally triggered or intentionally exploited to cause a failure [12]. Every vulnerability is one or more weaknesses. A weakness is not a vulnerability if it is guarded by code or has other mitigations anywhere in the broader system. For example, suppose an analyst is considering a dozen lines of code and sees that that piece of code has no protection from an SQL Injection attack. The code has an SQL Injection weakness. However, if the broader system context filters out any possible string with SQL Injection attacks, there is no vulnerability. We distinguish a weakness from a vulnerability to focus on a manageable context around pieces of code without constantly facing technically undecidable considerations like, "... if this code is reachable" or "... unless this filter is perfect".

For the first evaluation, that is, during SATE V, we wanted to be very precise about evaluating tool warnings. To be precise, we used the concept of *site*. A site "is a location in code where a weakness might occur." [9, Sec. 2.2]. In other words, sites of a weakness are places that must be checked for that weakness. For example, every buffer access in a C program is a site where buffer overflow might occur if the code is buggy. See Ref. [9, Sec. 2.2] for further exposition of what constitutes a site.

For each class of weakness, we found *all* sites then checked that there were no findings at those sites which had bugs [9, Sec. 3.1]. We spent much time and effort precisely matching our and Frama-C's definition of each class and the sites for them. For example, Frama-C only checked for signed arithmetic overflows for types of width `int` or larger.

Following are details of our approach for SATE V Ockham. We developed a program to extract all sites of all classes of warnings. This extractor also determined which sites were buggy by examining the source code and comments in it, as noted above.

For each warning class, we selected the universe, U, of corresponding sites, both those with bugs and those without bugs. If a warning, $w \in W$, was not in U, $w \notin U$, we debugged and corrected our extraction code or changed our classification of warnings so that there was an exact match between classes reported by Frama-C and our concept of classes. We then computed the findings, F, as sites without any warning, formally, $F = U - W$. If a class had any buggy sites, B, that were not in U, that is $B \not\subseteq U$, we also corrected our extraction code. The final check is that no finding was a buggy site, formally $F \cap B = \emptyset$. This satisfies Ockham criteria 3. (Criterion 3 is explained in Sec. 2.3.) The relations between these is illustrated in Fig. 1.

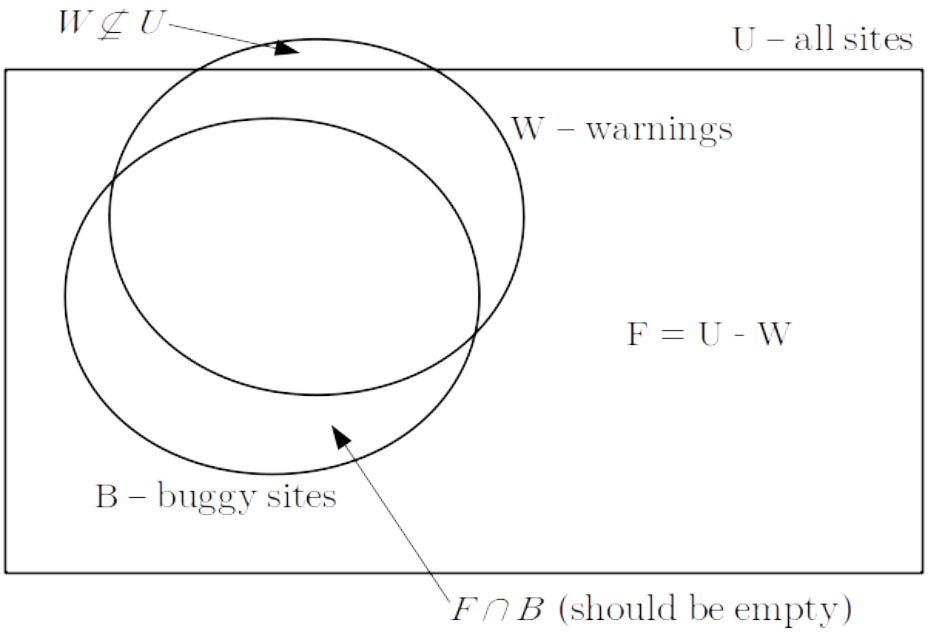

Fig. 1. Relation between the universe of sites, U, warnings reported, W, known buggy sites, B, and findings, F, computed as sites without warnings for SATE V Ockham. A crosscheck is that $W \subseteq U$. Ockham criterion 3 is satisfied if $F \cap B$ is empty.

After review, we decided that determining sites for each warning class was not worth the effort. In the current Ockham, SATE VI, we skipped the step of defining sites. We simply checked that all buggy sites were included in warnings, that is, $B \subseteq W$, see Fig. 2. Analytically, this is exactly the same check as $F \cap B = \emptyset$, with no need to compute the universe of sites.

Not determining sites means that we cannot calculate that Ockham criterion 2, explained in Sec. 2.2, is satisfied.

What risk of incorrect evaluation do we have when we do not determine sites? There are two possible mistakes: extraneous tools warnings included in a class and buggy sites

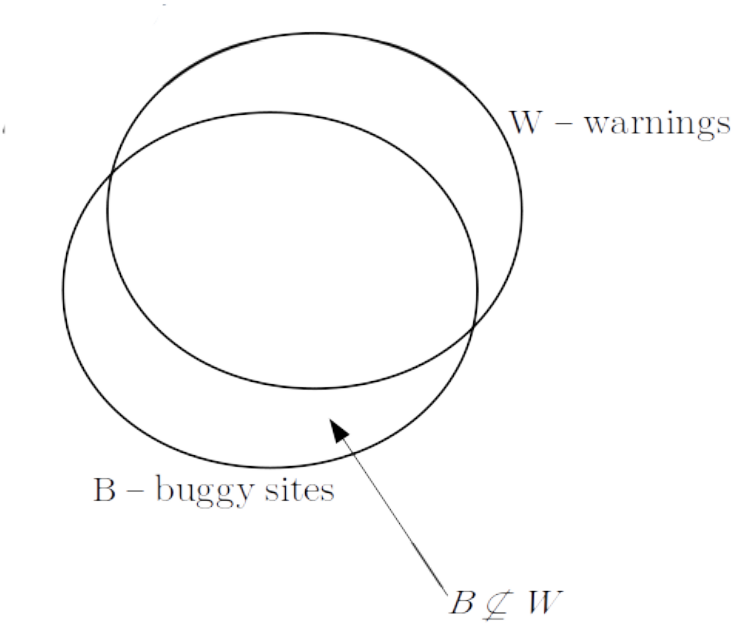

Fig. 2. Relation between warnings reported, W, and known buggy sites, B, for SATE VI Ockham. Ockham criterion 3 is satisfied if $B \subseteq W$.

not included in a class.

It is possible that some tool warnings are incorrectly assigned to a class and these extraneous warnings are matched with buggy sites that otherwise would be unmatched. Determining sites separately reduces the chance of extraneous warnings.

We minimize this risk by using a single, common program to assign all warnings to classes. If a warning was included in one class and should have been included in another class, checking the other class reveals that mistake. Checking different classes often revealed that warnings were assigned to the wrong class.

The other possible mistake is that buggy sites are not included in a class but should have been. Precisely determining sites reduces the chance of missing buggy sites.

Since we use a manifest of buggy sites that are labeled by CWE and CWEs generally correspond to the classes we chose, it is unlikely that buggy sites will be overlooked.

1.2.3 Bug or Weakness Classes

The Juliet test cases are grouped by Common Weakness Enumeration (CWE) [13]. For SATE V Ockham, we generally followed those classes. It was often difficult to assign tool warnings as one of those classes. We believed the difficulty was due to the lack of precision and detail in CWE class definitions.

For SATE VI Ockham, we defined our own classes to be orthogonal (no overlaps) and precise, following the approach of the Bugs Framework [14, 15]. Our classes are listed and defined in Sec. 2.4.

We still had significant difficulty assigning tool warnings to these new classes. Part of the problem was understanding *exactly* what the tool warning covered or what it did not cover. Part of the problem was that our classes made distinctions that tools did not and vice versa. Our own classes were easier to use than CWEs, but did not come close to being reasonable universal classes.

2. The Criteria

This section has the details of the Ockham Criteria itself, and includes explanation and discussion. Much of this section comes from Sec. 2 of the SATE V Ockham report [9].

The Criteria is named for William of Ockham, best known for Occam's Razor. Since the details of the Criteria will likely change in the future, the name includes a time reference: SATE VI Ockham Sound Analysis Criteria.

The value of a sound analyzer is that every one of its findings can be assumed to be correct.

We tried to write criteria that communicated our intent, ruled out trivial satisfaction, and were understandable. We planned to be liberal in interpreting the rules: we anticipated that tools satisfy the Criteria, so we occasionally assumed proper operation of the tool in cases requiring human judgment.

The criteria were:

1. The tool is claimed to be sound.
2. For at least one weakness class and one test case the tool produces findings for a minimum of 75 % of appropriate sites.
3. Even one incorrect finding disqualifies a tool for this SATE.

An implicit criterion is that the tool is useful, not merely a toy.

A *finding* is a definitive report about a site, which is a specific place in code. In other words, the tool reports that the site has a specific weakness (is buggy) or that the site does not have that weakness.

No manual editing of the tool output was allowed. No automated filtering specialized to a test case or to SATE VI was allowed, either. The tool's settings and options may be selected to produce the best result, as alluded to in Sec. 1.1. Such setting should be reported.

2.1 Criterion 1: "Sound" (and "Complete") Analysis

Criterion 1 is "The tool is claimed to be sound."

We use the term *sound* to mean that every finding is correct. In other words, "Sound analysis means that the [tool] never asserts a property to be true when it is not true." [16, FM.1.6.2]. The tool need not produce a finding for every site; that is completeness.

A tool may have settings that allow unsound analysis. The tool still qualifies if it has clearly sound settings. For example, it is acceptable for the user to be able to select unsound

(approximate) function signatures for fast analysis. A more inclusive statement of Criterion 1 is, the tool is claimed to be sound or has a mode in which analysis is sound.

A sound analyzer may produce false positives, that is, incorrectly report a problem when there is none. Consider checking for a divide by zero failure in the following code fragment:

```
int x = readInput();
if (x != 0) {
    x = 1776/x;
}
```

Suppose analysis tracks possible values of a variable as a range of values from a minimum to a maximum. After the first line, `x` can have any `int` value. This can be represented exactly as a range from the minimum `int` to the maximum `int`. Immediately after the conditional test, the possible values of `x` is all values except zero, which cannot be precisely represented as a single range. To always report possible bugs, the analyzer must overapproximate and continue to represent possible values as the entire range. When analysis checks the next line, zero is in the range of possible values. Analysis reports a (possible) divide by zero, even though it cannot actually occur.

For a more detailed exposition of uses of the terms "sound" and "complete" applied to static analysis, see Ref. [9, Sec. 2.3].

2.2 Criterion 2: Tools Produce Findings for Most Sites

Criterion 2 balances usefulness with theoretical limits: the tool produces findings for a minimum of 75 % of sites.

A trivial tool could produce no findings at all, which is arguably not incorrect. But a useful tool must produce findings for many sites in many pieces of software. Then why not require that a tool reports *all* buggy sites with no false positives?

This is impossible in theory. This impossibility often arises in practice, too. Rice's Theorem states that for any nontrivial property *any* algorithm either fails to report some cases when the property is present, or incorrectly reports the property's presence when it is absent [17].

A sound tool might report three cases: sites that definitely have a certain weakness, sites that definitely do not have a certain weakness, and sites for which the tool cannot determine.

After consultation with the SATE program committee, we chose 75 % as a level that is useful, yet readily achievable by current tools. In the future, we will likely set a higher limit.

The phrase, "For at least one weakness class ..." welcomes tools that focus on particular weakness classes, for instance, memory safety, reachability, or concurrency.

Processing different classes of weaknesses may take very different algorithmic machinery. The models, abstractions, data structures, and algorithms to look for one weakness may be of little help for another weakness.

Instead of trying to determine some required set of weaknesses, we allowed those running tools to designate the weakness or weaknesses that the tool finds and to choose one or more test cases.

The phrase, "and [at least] one test case" welcomes tools that require too many resources to analyze millions of lines of code. Juliet cases are so small that at least one whole CWE category must be handled to satisfy this phrase.

As explained previously, a *site* "is a location in code where a weakness might occur." [9, Sec. 2.2].

A *finding* may be that a site is buggy or that a site does not have a particular bug. Either type of statement (or both!) is acceptable. For instance, a tool may use conservative approximations and sometimes produce warnings about (possible) bugs at sites that are actually bug free. If it never misses a bug, then any site without a warning is sure to be correct. The tool makers could declare that sites without warnings are findings, and that all findings are correct.

Equivocal reports like "this site is likely to have that weakness," "caution: this function does not check for a null," or "this site is almost certainly safe" are at best ignored (not counted) and may be considered incorrect.

Because we did not determine sites for weakness classes, we cannot calculate the criterion 2 is satisfied.

2.3 Criterion 3: Determining That All Findings Are Correct

Criterion 3 is "Even one incorrect finding disqualifies a tool for this SATE." This section describes the general procedure we followed to confirm that a tool satisfied Criterion 3.

Initial comparison between findings and the Juliet manifest almost always produced thousands of mismatches.

One reason for mismatches is that reasoning is based on models, assumptions, definitions, etc. (collectively, "models"). Mismatches that result from model differences do not automatically disqualify a tool. In consultation with the tool maker, we decided if an unexpected finding resulted from a reasonable model difference or whether it was incorrect. To satisfy the SATE VI Ockham Criteria, any such differences are publicly reported.

For instance, one tool may assume that file system permissions are set as the test case requires, while another tool assumes the worst case about permissions. In this case, the tools could have different findings, yet both satisfy the Criteria. However, if a tool modeled "+" as subtraction or sometimes incorrectly modeled recursive calls, it was incorrect.

We performed the bulk of the analysis with automated scripts and custom programs. Automated scripts allowed us to rerun with relative ease as needed. Some exclusions and special handling were built into the code. These are mentioned in relevant sections. The steps are listed below and are given in Fig. 3.

1. Distill bugs from the list of known bugs in the test cases.
2. Run the tool on the test cases.
3. Extract findings from the tool results.

4. Check that all bugs are in the findings.
    - If so, Criterion 3 is satisfied.

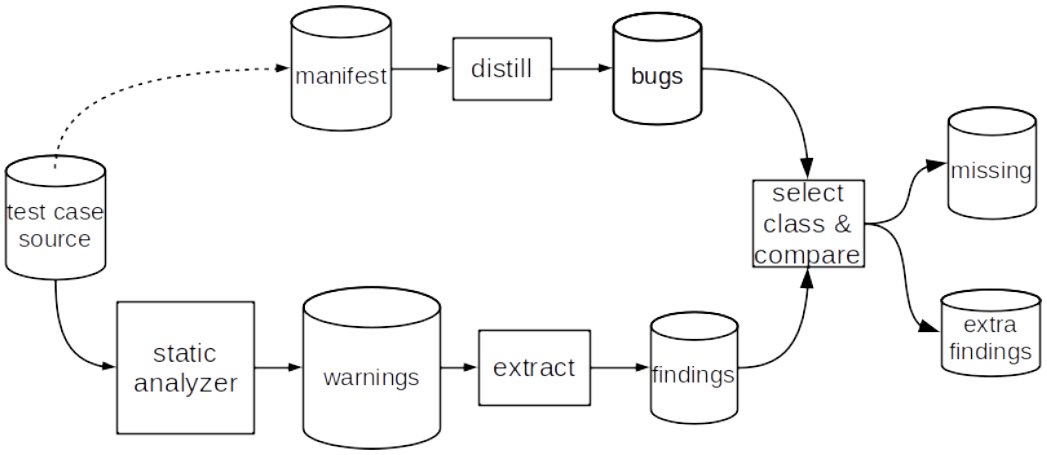

Fig. 3. General flow to confirm that a tool satisfied the SATE VI Ockham Sound Analysis Criterion 3. Distill bugs from the known-bugs manifest. Run the tool on the test cases. Extract findings from the tool output. Compare bugs and findings.

We produced the manifest of known bugs in Juliet 1.3 test cases by finding patterns in the comments and the code, by manual inspection, and by reconciliation with tool findings.

When we distill bugs and extract findings, we encode them as classes that we created for a common representation. The classes are listed and explained in Sec. 2.4.

We usually examined a tool's warnings one class at a time.

When a bug from the known list is missing from findings extracted from the tool result, we generally use the following steps to resolve the discrepancy.

1. Check that the bug is in the source code at the given location and of the indicated class. A bug not in the source code or of a different class may indicate an error in the manifest.
2. Check that the bug is in the manifest correctly.
3. Check that the bug is in the distilled bugs correctly. A problem suggests an error in the manifest extractor.
4. Check that a warning for the bug is produced by the tool. A problem may indicate
    - The tool was not run on that test case at all. This may be an error in a run script or a human oversight.
    - The tool should be run with different settings or with supplemental files.
    - The tool was not even programmed to detect this bug class.
    - The tool does not warn about this bug class because of something in the test case.

    The tool maker may provide insight.
5. Check that the bug is in the extracted findings correctly. A problem suggests an error in the warning extractor. The error could be that the extractor does not recognize the

warning, does not extract the information correctly, or encodes the warning as the wrong class.

6. If the bug is in the list of known bugs and in the extracted findings, the problem may be in the selection or comparison. In addition, one of the files may be out of date and programs needs to be run again.

Sometimes we examined the extra findings that do not match any of the bugs in the manifest to minimize our errors. An unmatched finding may indicate valid bugs missing from the manifest, findings that are incorrectly encoded, or findings that are not extracted at all. For those, we adapt the steps from above. They may be actual false positives, which are allowed in the Ockham Criteria.

Some mismatches may be caused by our choice of bug classes. For example, we check write buffer overflows (BOF/Write) separately from read buffer overflows (BOF/Read). A tool may only warn that the code is accessing an index out of bounds and not indicate whether it is a read or write. In these cases, we may use context, like the test case name, to classify the warning.

2.4 Ockham Bug Classes

We defined our own classes to organize our examination of warnings. We tried to follow the approach of the Bugs Framework [14] to precisely and rigorously define clear classes. The classes must not overlap. That is, a bug must be in one class or another.

In many instances, our classes did not correspond well to the tool warnings.

This section explains each class. We treat closely related classes or subclasses in a single subsection.

2.4.1 ARC—Arithmetic or Conversion Fault Classes

We begin with the Arithmetic or Conversion Fault, or ARC, class. For the SATE VI Ockham Criteria, we define ARC as

> Software produces a faulty result due to conversions between primitive types, range violations, or domain violations.

Range violation comprises truncation, overflow, underflow, or distortion. We categorize "wrap around" as either overflow or underflow.

Because there are so many ARC test cases of different kinds in Juliet, we subdivided ARC into several classes: Truncate, Overflow, Underflow, Distort, and Undefined.

ARC/Truncate is when the value to be stored is too big for the destination and is cut off, e.g.,

```
int i1 = 12345678901L;
```

Test cases for this class are under CWE197_Numeric_Truncation_Error.

ARC/Overflow is when the types match, but the result is too big for the destination, e.g.,

```
unsigned int i2 = UINT_MAX + 1;
```

Most test cases for this class are under CWE190_Integer_Overflow and CWE680_ Integer_Overflow_to_Buffer_Overflow. A few are under CWE196_Unsigned_to_Signed_ Conversion_Error.

ARC/Underflow is when the types match, but the result is too negative for the destination, e.g.,

```
int i3 = INT_MIN - 1;
```

Test cases for this class are under CWE191_Integer_Underflow.

ARC/Distort is when the result is otherwise distorted, e.g.,

```
unsigned i4 = -5;
```

Test cases for ARC/Distort are under CWE194_Unexpected_Sign_Extension and CWE195_Signed_to_Unsigned_Conversion_Error.

To contrast the four preceding subclasses, truncate is when the type of the source is larger than the type of the destination. Overflow and Underflow are when the destination is at least as big as the source, but the result of an arithmetic operation won't fit in the destination type. Distort is when the destination is at least as big as the source and the result would fit, but it is changed for another reason.

ARC/Undefined is a domain violation, such as divide by zero or negative shift.

Test cases for this class are under CWE369_Divide_by_Zero.

2.4.2 ARG/Memcpy—Incorrect Argument for `memcpy()`

We define ARG as

> Software calls a function with incorrect arguments.

Test cases for this class are under CWE475_Undefined_Behavior_for_Input_to_API. They copy overlapping memory areas with `memcpy()`. The following example is from `CWE475_Undefined_Behavior_for_Input_to_API__char_01.c` SARD case 104 496

```
memcpy(data + 6, data + 4, 10*sizeof(char));
```

In future Ockham evaluations, we may use additional test cases under CWE685_Function_Call_With_Incorrect_Number_of_Arguments. The test cases call `sprintf()` with the wrong number of arguments. The following example is from `CWE685_Function_Call_With_Incorrect_Number_of_Arguments__basic_02.c`, SARD case 110 943

```
sprintf(dest, "%s %s", SOURCE_STRING);
```

2.4.3 BOF/Read and BOF/Write—Read or Write Outside Buffer

We define BOF as

> Software accesses through an array a memory location that is outside the boundaries of that array.

This definition is after [14].

Because there are so many BOF bugs in Juliet, we divided this into BOF/Read and BOF/Write classes. Warnings may not indicate whether it was a read or write; only that a BOF may occur. In these cases, we use information about which test case yielded the warning to encode it as BOF/Read or BOF/Write.

Test cases for BOF/Read are under CWE126_Buffer_Overread and CWE127_Buffer_Underread. Most test cases for BOF/Write are under CWE121_Stack_Based_Buffer_ Overflow, CWE122_Heap_Based_Buffer_Overflow, CWE124_Buffer_Underwrite, and CWE680_Integer_Overflow_to_Buffer_Overflow. A few test cases are under CWE242_Use_of_ Inherently_Dangerous_Function, CWE665_Improper_Initialization, and CWE685_Function_Call_With_Incorrect_Number_of_Arguments.

2.4.4 DEP—Dereference Erroneous Pointer Classes

We define DEP as

> Software dereferences an invalid pointer.

The invalid pointer may be `NULL`, refer to memory that was freed (see MAL below), be produced by incorrect pointer arithmetic (see PAR below), or be completely arbitrary.

Because Juliet has particular cases for incorrect pointer arithmetic, we separate them into their own class, DEP/Incorrectly Computed Pointer (DEP/ICP). Test cases for this class are under CWE188_Reliance_on_Data_Memory_Layout and CWE588_Attempt_to_Access_Child_of_Non_Structure_Pointer.

Test cases for other DEP faults are under CWE476_NULL_Pointer_Dereference, CWE690_NULL_Deref_From_Return, and CWE123_Write_What_Where_Condition.

2.4.5 PAR—Pointer Arithmetic

We define PAR as

> Software produces a faulty value because of incorrect pointer arithmetic.

DEP is when an incorrect pointer is dereferenced. PAR is when the result of pointer arithmetic is used as a number or when pointers are compared incorrectly.

Test cases for this class are under CWE253_Incorrect_Check_of_Function_Return_Value (only _fgets_) and CWE469_Use_of_Pointer_Subtraction_to_Determine_Size.

Test cases under CWE467_Use_of_sizeof_on_Pointer_Type and CWE468_Incorrect_Pointer_Scaling may be useful in the future.

2.4.6 ILP—Infinite Loop

We define ILP as

> Software never terminates execution.

Test cases for this class are under CWE835_Infinite_Loop.

2.4.7 INI—Initialization Fault

We define INI as

> Software uses a faulty value because an entity was not properly initialized.

"Entity" includes a variable, a member of a structure or record, parts of an array, a pointer or reference, etc. "Not properly" covers both not initialized at all and not correctly initialized. The entity may not be initialized at all for many reasons: the programmer neglected to write initialization at all, program execution followed an unexpected path, the initialization routine was not called, etc. Not initialized correctly includes initialized with a value that leads to a security concern.

Most test cases for this class are under CWE457_Use_of_Uninitialized_Variable. A few are under CWE665_Improper_Initialization.

2.4.8 MAL—Memory Allocation and Deallocation

We define MAL as

> Software improperly allocates or deallocates memory.

Test cases for this class are under CWE415_Double_Free, CWE416_Use_After_Free, and CWE761_Free_Pointer_Not_at_Start_of_Buffer.

We also include memory leaks in this class. However Juliet does not have explicit test cases for memory leak, either failure to free or losing any pointer to allocated memory.

2.4.9 UCE—Unchecked Error

We define UCE as

> Software does not check, checks incorrectly, or checks but not take action on a possible error condition.

Test cases for this class are under CWE252_Unchecked_Return_Value.

Additional test cases for this class are under CWE390_Error_Without_Action and CWE391_Unchecked_Error_Condition.

3. SATE VI Evaluation

We evaluated several tools by the SATE VI Ockham Sound Analysis Criteria. This section has one subsection for each tool.

All of the scripts and files are available in a tar file with xz compression [18] at DOI http://dx.doi.org/10.18434/M32187 or https://nist-sate-ockham-sound-analysis-criteria-evaluation-material.s3.amazonaws.com/ockham-sate-VI-2020/ockhamCriteriaSATEVI data2020.tar.xz The README is available at https://nist-sate-ockham-sound-analysis-criteria-evaluation-material.s3.amazonaws.com/ockham-sate-VI-2020/README

3.1 Astrée

"Astrée is a static code analyzer that proves the absence of run-time errors and invalid concurrent behavior in safety-critical software written or generated in C. . . . Astrée is sound for floating-point computations and handles them precisely and safely. . . . Astrée offers powerful annotation mechanisms for supplying external knowledge and fine-tuning the analysis precision for individual loops or data structures. . . . This allows for analyses with very few or even zero false alarms." [5]

AbsInt Angewandte Informatik GmbH granted NIST an evaluation license to run Astrée for C on a Linux 64-bit platform. In June 2018 we installed a^3 for C 18.04 (2819232) and began analyzing the Juliet test cases.

By its own definition, Astrée claimed to be sound: "Astrée is sound — that is, if no errors are signaled, the absence of errors has been proved." [5]. This satisfies Criterion 1.

Since Astrée produced thousands of warnings, we believe it would satisfy Criterion 2.

3.1.1 Performing the Evaluation

What we refer to elsewhere in this report as warnings, Astrée refers to as alarms. We used the classes listed in Sec. 2.4 to organize our examination of Astrée alarms. We examined alarms generally class by class.

AbsInt supplied the initial wrappers and stub code, declaration and configuration (.dax) files, and Astrée Annotation Language (.aal) files. AbsInt notes that the stubs are only example implementations and should always be checked to determine what enhancements, if any, are necessary to match the libraries in use and the properties of importance.

We ran Astrée on the following Juliet 1.3 sets of test cases:

- CWE121_Stack_Based_Buffer_Overflow
- CWE122_Heap_Based_Buffer_Overflow
- CWE123_Write_What_Where_Condition
- CWE124_Buffer_Underwrite
- CWE126_Buffer_Overread
- CWE127_Buffer_Underread
- CWE188_Reliance_on_Data_Memory_Layout
- CWE190_Integer_Overflow
- CWE191_Integer_Underflow
- CWE194_Unexpected_Sign_Extension
- CWE195_Signed_to_Unsigned_Conversion_Error
- CWE196_Unsigned_to_Signed_Conversion_Error
- CWE197_Numeric_Truncation_Error
- CWE242_Use_of_Inherently_Dangerous_Function
- CWE252_Unchecked_Return_Value
- CWE253_Incorrect_Check_of_Function_Return_Value
- CWE364_Signal_Handler_Race_Condition
- CWE366_Race_Condition_Within_Thread

- CWE367_TOC_TOU
- CWE369_Divide_by_Zero
- CWE398_Poor_Code_Quality
- CWE404_Improper_Resource_Shutdown
- CWE415_Double_Free
- CWE416_Use_After_Free
- CWE457_Use_of_Uninitialized_Variable
- CWE475_Undefined_Behavior_for_Input_to_API
- CWE476_NULL_Pointer_Dereference
- CWE478_Missing_Default_Case_in_Switch
- CWE561_Dead_Code
- CWE587_Assignment_of_Fixed_Address_to_Pointer
- CWE588_Attempt_to_Access_Child_of_Non_Structure_Pointer
- CWE665_Improper_Initialization
- CWE667_Improper_Locking
- CWE680_Integer_Overflow_to_Buffer_Overflow
- CWE685_Function_Call_With_Incorrect_Number_of_Arguments
- CWE690_NULL_Deref_From_Return
- CWE761_Free_Pointer_Not_at_Start_of_Buffer
- CWE832_Unlock_of_Resource_That_is_Not_Locked
- CWE835_Infinite_Loop

We extracted the following alarm names for matching:

- Arithmetics on invalid pointers
- Dereference of null or invalid pointer
- Float division by zero
- Function return unused
- Incorrect field dereference
- Infinite loop
- Integer division by zero
- Integer modulo by zero
- Integer overflow
- Invalid argument in dynamic memory allocation, free or resize
- Invalid pointer comparison
- Memcpy overlapping
- Out-of-bound array access
- Overflow in arithmetic
- Overflow in conversion
- Overflow in conversion (with unpredictable result)
- Possible overflow upon dereference
- Reinterpreting incompatible function return type
- Reinterpreting incompatible parameter type in a function call
- Uninitialized local read

- Use of dangling pointer
- Use of uninitialized variables
- User defined alarm

Astrée allows users to enable or disable rules and checks individually or as groups.

3.1.2 Common Considerations

This section explains some considerations that applied to all the classes. These are details of Astrée operation or steps we took to match Astrée alarms to our notion of classes, locations, and warnings.

In many cases the line that Astrée produces and the line in the manifest are different, but both are reasonable. Consider the following code, which comes from `CWE835_Infinite_Loop__do_01.c` SARD case 122 761:

```
do
{
    printIntLine(i);
    i = (i + 1) % 256;
} while(i >= 0);
```

Astrée issues an alarm at the start of the loop. The manifest lists the error at the end of the loop. In these cases the program that extracts alarms from Astrée output, what we call the *extractor*, `astreeXML2csv`, patches the extracted alarm to correspond with the manifest. See Sec. 3.1.14.

Another example of different line numbers is illustrated by this code from `CWE122_Heap_Based_Buffer_Overflow__sizeof_struct_01.c` SARD case 234 204 is:

```
typedef struct
{
    int intOne;
    int intTwo;
} twoIntsStruct;

23    twoIntsStruct * data;

28    data = (twoIntsStruct *)malloc(sizeof(data));
29    if (data == NULL) {exit(-1);}
30    data->intOne = 1;
31    data->intTwo = 2;
```

The problem is that at line 28, only memory for a *pointer* to the `struct` is allocated, not memory for the whole `struct`. Astrée determines that line 31 is undefined because the assignment is beyond the amount of memory allocated. It does not issue an alarm for line 30 because it is well defined. The manifest incorrectly lists an error at line 30 and should be fixed. As above the extractor, `astreeXML2csv`, patches the extracted alarm to correspond with the manifest. See Sec. 3.1.10.

Here is yet another case, which comes from `CWE690_NULL_Deref_From_Return__struct_malloc_01.c` SARD case 111 650:

```
      data = (twoIntsStruct *)malloc(1*sizeof(twoIntsStruct));
30    data[0].intOne = 1;
31    data[0].intTwo = 1;
```

The manifest lists (possible) NULL pointer dereference at both lines 30 and 31. Astrée issues an alarm only about the first line. The extractor adds a second alarm to match the manifest. See Sec. 3.1.11.

Sometimes Astrée reports an alarm as in a utility file, not in the calling function, where it would be convenient for us. With highly automated processing, we added some context-sensitive checks.

Other considerations are that Astrée did not support C++, so we excluded all `.cpp` cases[1]. Astrée needs library stubs and type definitions to support Windows-specific code. Since we did not have them, we excluded `_w32_` and `_wchar_t_` test cases. Astrée followed C99, not C11, semantics.

Following is one subsection for each weakness class. We include details about the evaluation of a class when useful.

3.1.3 ARC/Overflow—Arithmetic Overflow

Anomalies, Observations, and Interpretations:

Analyzing this class showed that the manifest was missing ARC/Overflow bugs for CWE680 cases. Here is pertinent code from `CWE680_Integer_Overflow_to_Buffer_Overflow__malloc_rand_01.c` SARD case 241 054:

```
33      intPointer = (int*)malloc(data * sizeof(int));
        for (i = 0; i < (size_t)data; i++)
        {
37          intPointer[i] = 0;
        }
```

At line 33, `data * sizeof(int)` may exceed an integer and overflow, an ARC/Overflow. Not enough memory is allocated, causing a BOF at line 37, which the manifest has. It did not have an ARC/Overflow at line 33. We added the 576 ARC/Overflow bugs to the manifest.

Results:

3666 buggy sites.

All buggy sites were found.

For ARC/Overflow, Astrée satisfied Ockham Criterion 3.

[1] C++ support has been added and is scheduled to be available to all users with the 20.04 release.

3.1.4 ARC/Underflow—Arithmetic Underflow

Anomalies, Observations, and Interpretations:

The vast majority of the unmatched findings (4412 of 4792) are uses of the `RAND32` macro. The rest (380) are code with `something-1`, which underflows for some types.

We found poor code in "good" functions in CWE191. Here is an example from `CWE191_Integer_Underflow__unsigned_int_fscanf_postdec_01.c` SARD case 237 917

```
      unsigned int data;
        ...
      data = -2;
        ...
30    data--;
```

Although the assignment avoids underflow at the decrement, line 30, the "fix" should assign a positive value, e.g., `data = 2;`.

We also found that "bad" functions in CWE680 had extraneous errors. Pertinent code from `CWE680_Integer_Overflow_to_Buffer_Overflow__malloc_fscanf_01.c` SARD case 240 972 is

```
      fscanf(stdin, "%d", &data);
      {
        ...
        intPointer = (int*)malloc(data * sizeof(int));
```

The purpose of this code is that `data` could be a very large value, leading to integer overflow because of the multiplication. This is fine. However, `data` could also be negative. This is an unintended ARC/Distort. The code could be fixed by adding a guard like this:

```
      if (data > 0) {
        ...
          intPointer = (int*)malloc(data * sizeof(int));
```

Results:

2622 buggy sites.
All buggy sites were found.

For ARC/Underflow, Astrée satisfied Ockham Criterion 3.

3.1.5 ARC/Undefined—Divide by Zero

Results:

684 buggy sites.
All buggy sites were found.

For ARC/Undefined, Astrée satisfied Ockham Criterion 3.

3.1.6 ARC/Distort—Result Distortion

Results:

1824 buggy sites.
All buggy sites were found.

For ARC/Distort, Astrée satisfied Ockham Criterion 3.

3.1.7 ARC/Truncate—Result Truncation

Anomalies, Observations, and Interpretations:

Initially we had many ARC/Truncate alarms from cases under CWE367_TOC_TOU and CWE404_Improper_Resource_Shutdown. We posit that `open()` is modeled as returning `signed long long`, which would be truncated (ARC/Truncate) to fit in int. However, Ref. [19] says `open()` returns `int`, so the Juliet code is not buggy.

Many alarms for cases under CWE369_Divide_by_Zero are valid, but not interesting to us. Here is pertinent code from `CWE369_Divide_by_Zero__float_fgets_13.c` SARD case 94 734:

```
    data = (float)atof(inputBuffer);
```

Since `atof()` returns a `double`, this is an ARC/Truncate.

We did not have a .dax files for CWE681_Incorrect_Conversion_Between_Numeric_Types, so we did not run Astrée on those cases. Analysis did not select those from the manifest.

Initial comparison showed 26 buggy sites not in the findings. All of them were flow variants of `CWE197_Numeric_Truncation_Error__short_fscanf_*`. For example, in `CWE197_Numeric_Truncation_Error__short_fscanf_01.c` SARD case 89 280, the manifest had an ARC/Truncate at line 27. The pertinent code is:

```
    short data;
       ...
27  fscanf (stdin, "%hd", &data);
```

This code is does not have a bug. We removed the 26 incorrect entries from the manifest.

Results:

684 buggy sites.
All buggy sites were found.

For ARC/Truncate, Astrée satisfied Ockham Criterion 3.

3.1.8 ARG/Memcpy—Incorrect Argument for `memcpy()`

Results:

18 buggy sites.
All buggy sites were found.

For ARG/Memcpy, Astrée satisfied Ockham Criterion 3.

3.1.9 BOF/Read—Read Outside Buffer

Anomalies, Observations, and Interpretations:

Reconciling (the lack of) Astrée alarms found 72 "fossil" errors in the manifest. The Juliet code had been corrected in Juliet 1.3, see [11, Sec. 2.3], but the errors hadn't been removed from manifest. These were incidental errors in CWE121 and CWE122 cases where a source string for a memory copy or move was too short for 64 bit architectures.

All 1450 unmatched alarms are in code calling `printLine()`, similar to this, which comes from `CWE121_Stack_Based_Buffer_Overflow__CWE135_01.c` SARD case 231 402:

```
37      (void)wcscpy(dest, data);
        printLine((char *)dest);
```

Preceding code does not allocate enough memory for `dest` to hold the wide string copied from `data`. Therefore, `wcscpy()` writes a string that goes beyond `dest`. Since `printLine()` prints everything to the end of the string, it reads outside `dest`. Astrée warns about this read outside the bounds of `dest`. This is valid, but we decided not to add these to the manifest since these are cascading bugs and the out-of-bounds access is already listed for line 37.

Initially we got extraneous alarms about BOF/Read in code like `CWE126_Buffer_Overread__CWE170_char_loop_01.c` SARD case 75 886. The pertinent code is:

```
          char src[150], dest[100];
          memset(src, 'A', 149);
          for(i=0; i < 99; i++)
          {
32            dest[i] = src[i];
          }
```

The alarms were "uninitialized read: reading 1 byte(s) at offset(s) nn in variable src" at line 32. We learned that Astrée does not, by default, unroll loops sufficiently to notice that enough is initialized. Increasing the loop unrolling permitted proper analysis.

Initially we were missing 54 BOF/Read alarms. An example is in code from `CWE126_Buffer_Overread__CWE170_char_loop_01.c` SARD case 75 886, line 35:

```
          char src[150], dest[100];
          memset(src, 'A', 149);
          for(i=0; i < 99; i++)
          {
              dest[i] = src[i];
          }
          // FLAW: dest is not null terminated.
35        printLine(dest);
```

This was because the model of "printf() is treated as an empty stub". We replaced the model with the following body suggested by Christoph Mallon:

```
__ASTREE_unroll((200))
for (char const* p = line; *p != '\0'; ++p) {}
```

This yielded matching alarms.

Results:
1188 buggy sites.
All buggy sites were found.

For BOF/Read, Astrée satisfied Ockham Criterion 3.

3.1.10 BOF/Write—Write Outside Buffer

Anomalies, Observations, and Interpretations:

Astrée produced "invalid dereference" alarms for the call to `strlen()` in cases like the following from `CWE121_Stack_Based_Buffer_Overflow__CWE135_01.c` SARD case 231 402:

```
#define WIDE_STRING L"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
  ...
    data = (void *)WIDE_STRING;
      ...
        size_t dataLen = strlen((char *)data);
```

Even though the code incorrectly uses `strlen()` on a wide string, it is not BOF/Write. Given more time, we would have considered improving the model of `strlen()` to match our library.

The manifest has warnings for the `memcpy()` in cases such as `CWE121_Stack_Based_Buffer_Overflow__char_type_overrun_memcpy_01.c` SARD case 231 444:

```
typedef struct _charVoid
{
    char charFirst[16];
    void * voidSecond;
    void * voidThird;
} charVoid;

        charVoid structCharVoid;
                    ...
        memcpy(structCharVoid.charFirst, SRC_STR,
                                        sizeof(structCharVoid));
```

The `memcpy()` writes beyond the `charFirst` buffer because the size of the entire structure is used, not the size of the buffer. `memcpy()` is often used to copy entire structures. We believe that Astrée gives no alarm for these because the data copied stays within the structure, although it goes outside the buffer. We filter out these unmatched manifest warnings.

For the previous Ockham Criteria, we added entries to the manifest for the possible null pointer dereference (DEP) on line 51 below. This example code is from `CWE121_Stack_Based_Buffer_Overflow__CWE805_char_alloca_loop_12.c` SARD case 63 843.

```
    char * dataBadBuffer = (char *)ALLOCA(50*sizeof(char));
      ...
        data = dataBadBuffer;
        ...
        for (i = 0; i < 100; i++)
        {
49          data[i] = source[i];
        }
51      data[100-1] = '\0';
```

Astrée reports an alarm at line 49 and stops analysis with a "Definite runtime error", which is acceptable. We ignored the second warning, line 51, which was straight-forward since they were tagged CWE-787 (Out-of-bounds Write).

We found that Astrée models `alloca()`[2] as possibly returning `NULL`. In cases such as those in `CWE127_Buffer_Underread__char_alloca_cpy_01.c` SARD case 77 242, the pertinent code is

```
26    char * dataBuffer = (char *)ALLOCA(100*sizeof(char));
27    memset(dataBuffer, 'A', 100-1);
30    data = dataBuffer - 8;
      ...
36        strcpy(dest, data);
```

The buffer are big enough, and `dataBuffer` is initialized correctly. But the read from `data` (this is BOF/Read, not BOF/Write) at line 36 begins before the beginning of the buffer. Because of Astrée's model of `alloca()`, it gives an alarm of possible NULL pointer dereference (DEP) at line 27 and stops analysis. We removed these `alloca()` cases from the buggy set to match.

Analyzing Astrée alarms led us to discover that the manifest had incorrect locations for many CWE665_Improper_Initialization cases. We corrected those in the manifest.

Results:
5192 buggy sites.
All buggy sites were found.

For BOF/Write, Astrée satisfied Ockham Criterion 3.

3.1.11 DEP—Dereference Erroneous Pointer

Anomalies, Observations, and Interpretations:

[2]The code uses a macro, `ALLOCA`, that just becomes `alloca`.

Some Juliet cases under CWE476_NULL_Pointer_Dereference are intended to find out if a tool reports a useless NULL check after a dereference. Here is pertinent code from `CWE476_NULL_Pointer_Dereference__null_check_after_deref_01.c` SARD case 104 778:

```
          intPointer = (int *)malloc(sizeof(int));
25        *intPointer = 5;

28        if (intPointer != NULL)
```

The check for a (non-)NULL pointer at line 28 is wasted at best. If the allocation fails and `intPointer` is NULL, the assignment at line 25 is faulty. Astrée produces alarms about possible NULL pointer dereference there. Since Astrée behaves reasonably, we skip these cases.

As with BOF/Write, Sec. 3.1.10, the model for `alloca()` led to many of the unmatched alarms, which we consider spurious.

Results:
1166 buggy sites.
All buggy sites were found.

For DEP, Astrée satisfied Ockham Criterion 3.

3.1.12 DEP/ICP—Incorrect Pointer Arithmetic

Anomalies, Observations, and Interpretations:

Below is pertinent code from `CWE188_Reliance_on_Data_Memory_Layout__union_01.c` SARD case 82 092:

```
    union {
        struct {
            char charFirst, charSecond, charThird, charFourth;
        } structChars;
        long longNumber;
    } unionStructLong;
    unionStructLong.longNumber = 0x10203040;
34  unionStructLong.structChars.charFourth |= 0x80; // set MSB
```

Line 34 depends on certain byte-order, size, alignment, and packing of struct and union fields. For our settings, Astrée doesn't produce any error for that line at all. In checking the Help manual, it appears that Astrée doesn't check for this at all. We therefore did not examine any CWE188 cases.

Results:
52 buggy sites.
All buggy sites were found.

For DEP/ICP, Astrée satisfied Ockham Criterion 3.

3.1.13 PAR—Pointer Arithmetic

Results:
18 buggy sites.
All buggy sites were found.

For PAR, Astrée satisfied Ockham Criterion 3.

3.1.14 ILP—Infinite Loop

Anomalies, Observations, and Interpretations:
We found a difference in reporting locations from code like the following, which comes from `CWE835_Infinite_Loop__do_01.c` SARD case 122 761:

```
do
{
    printIntLine(i);
    i = (i + 1) % 256;
} while(i >= 0);
```

Astrée issues an alarm at the start of the loop. The manifest lists the error at the end of the loop. In these cases the extractor, `astreeXML2csv`, patches the extracted alarm to correspond with the manifest.

Results:
6 buggy sites.
All buggy sites were found.

For ILP, Astrée satisfied Ockham Criterion 3.

3.1.15 INI—Initialization Fault

Anomalies, Observations, and Interpretations:
We found a difference in reporting locations from code like the following, which comes from `CWE457_Use_of_Uninitialized_Variable__char_pointer_63b.c` SARD case 103 357:

```
26    char * data = *dataPtr;

28    printLine(data);
```

Astrée warns when (a pointer to) an uninitialized variable is assigned on line 26. However, the uninitialized variable isn't used until line 28.

Initially there were about four times more unmatched alarms. We added directives to unroll much of the code with `memset()`, giving the number reported below. We believe that such directives could resolve most or all of them.

Results:
776 buggy sites.
All buggy sites were found.

For INI, Astrée satisfied Ockham Criterion 3.

3.1.16 MAL—Memory Deallocation

Anomalies, Observations, and Interpretations:

Evaluating Astrée lead us to discover that the manifest was wrong for code such as the following, which is from `CWE415_Double_Free__malloc_free_char_01.c` SARD case 240 071:

```
29    data = (char *)malloc(100*sizeof(char));
      ...
32    free(data);
      /* POTENTIAL FLAW: Possibly freeing memory twice */
34    free(data);
```

The manifest listed the first `free()`, at line 32, instead of the second one at line 34.

Resolving the discrepancy between lines also showed that the manifest was wrong for code such as the following, from `CWE761_Free_Pointer_Not_at_Start_of_Buffer__char_environment_01.c` SARD case 241 393. Prior to this, `data` is allocated and initialized.

```
52    for (; *data != '\0'; data++) {
          . . .
      }
60    free(data);
```

The manifest listed line 52, while the error is really at line 60.

Results:
536 buggy sites.
All buggy sites were found.

For MAL, Astrée satisfied Ockham Criterion 3.

3.1.17 UCE—Unchecked Error

Anomalies, Observations, and Interpretations:

We did not find that Astrée checked the handling of error returns from functions like `fread()`, `putchar()`, and `scanf()`, which are under CWE253. We therefore did not check any CWE253 cases.

We did not run Astrée on any cases under CWE390 or CWE391. Therefore we did not check any of those cases.

Results:
540 buggy sites.
All buggy sites were found.

For UCE, Astrée satisfied Ockham Criterion 3.

3.1.18 Summary of Evaluation

Alarms from Astrée led us to find and fix thousands of mistakes in what was intended as the Juliet known-bug list, manifest.xml.

Because Astrée analyzes code very precisely and we checked meticulously, details of modeling that otherwise would be inconsequential showed up and had to be resolved. For instance, our evaluation recognized minutia of models of `open()` (Sec. 3.1.7), `printf()` (Sec. 3.1.9), and `strlen()` and `alloca()` (Sec. 3.1.10).

In the test cases of the 28 sets used from the Juliet 1.3 C test suite, we considered 18 954 buggy sites for Astrée. Table 2 gives the number of buggy sites considered for each class. We processed a total of 36 316 Astrée alarms.

Class	Bugs
ARC_Distort	1824
ARC_Oflow	3666
ARC_Trunc	684
ARC_Uflow	2622
ARC_Undef	684
ARG_Memcpy	18
BOF_Read	1188
BOF_Write	5174
DEP	1166
DEP_ICP	52
ILP	6
INI	776
MAL	536
PAR	18
UCE	540

Table 2. Number of Buggy Sites Considered for Astrée for Each Weakness Class

Astrée satisfied the SATE VI Ockham Sound Analysis Criteria.

3.2 Frama-C

"Frama-C is a suite of tools dedicated to the analysis of the source code of software written in C." [6] "Frama-C allows [you] to verify that the source code complies with a provided formal specification. Functional specifications can be written in a dedicated language, ACSL. The specifications can be partial, concentrating on one aspect of the analyzed program at a time." [20] It is free software licensed under the GNU Lesser General Public License (LGPL) v2.1 license[3].

We began evaluation in June 2019 and used Frama-C 'Argon' 18.0 Version.

By its own definition, Frama-C claimed to be sound: "it aims at being *correct*, that is, never to remain silent for a location in the source code where an error can happen at run-time" [6]. This satisfies Criterion 1.

Since Frama-C with the Evolved Value Analysis (Eva) plug-in produced thousands of warnings, we believe it would satisfy Criterion 2.

3.2.1 Performing the Evaluation

To produce all the warnings we were interested in, we eventually ran Frama-C and Eva on each test case with four different sets of options. The first run only used `-eva-print-callstacks`. The second, third, and fourth runs all used this common set of options:

- `-eva-print-callstacks`
- `-eva-msg-key=-initial-state`
- `-eva-no-show-progress`
- `-slevel 300`
- `-warn-special-float none`
- `-eva-warn-signed-converted-downcast`
- `-warn-unsigned-overflow`
- `-eva-warn-copy-indeterminate=-@all`
- `-eva-equality-domain`
- `-eva-sign-domain`
- `-warn-signed-overflow`

The second run only used those options. The third run adds `-warn-signed-downcast` to the common set of options. The fourth run adds `-warn-unsigned-downcast` to the common set.

We checked the union of warnings from all four runs together for convenience. This decision lost precision. Results would be much clearer had we drawn particular warnings from runs with particular sets of options. For instance, guarded code that otherwise could divide by zero is analyzed properly using `-eva-sign-domain`. Hence, we should only have examined divide by zero warnings in runs using `-eva-sign-domain`.

We used the classes listed in Sec. 2.4 to organize our examination of Frama-C/Eva warnings. We examined warnings generally class by class.

[3] http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html

We ran Frama-C/Eva on the following Juliet 1.3 sets of test cases:

- CWE121_Stack_Based_Buffer_Overflow
- CWE122_Heap_Based_Buffer_Overflow
- CWE123_Write_What_Where_Condition
- CWE124_Buffer_Underwrite
- CWE126_Buffer_Overread
- CWE127_Buffer_Underread
- CWE190_Integer_Overflow
- CWE191_Integer_Underflow
- CWE194_Unexpected_Sign_Extension
- CWE195_Signed_to_Unsigned_Conversion_Error
- CWE196_Unsigned_to_Signed_Conversion_Error
- CWE197_Numeric_Truncation_Error
- CWE253_Incorrect_Check_of_Function_Return_Value
- CWE364_Signal_Handler_Race_Condition
- CWE366_Race_Condition_Within_Thread
- CWE367_TOC_TOU
- CWE369_Divide_by_Zero
- CWE398_Poor_Code_Quality
- CWE404_Improper_Resource_Shutdown
- CWE415_Double_Free
- CWE416_Use_After_Free
- CWE457_Use_of_Uninitialized_Variable
- CWE469_Use_of_Pointer_Subtraction_to_Determine_Size
- CWE475_Undefined_Behavior_for_Input_to_API
- CWE476_NULL_Pointer_Dereference
- CWE478_Missing_Default_Case_in_Switch
- CWE561_Dead_Code
- CWE587_Assignment_of_Fixed_Address_to_Pointer
- CWE588_Attempt_to_Access_Child_of_Non_Structure_Pointer
- CWE665_Improper_Initialization
- CWE667_Improper_Locking
- CWE680_Integer_Overflow_to_Buffer_Overflow
- CWE685_Function_Call_With_Incorrect_Number_of_Arguments
- CWE690_NULL_Deref_From_Return
- CWE761_Free_Pointer_Not_at_Start_of_Buffer
- CWE832_Unlock_of_Resource_That_is_Not_Locked

We extracted the following warning texts:

- accessing left-value that contains escaping addresses
- accessing uninitialized left-value
- division by zero
- implicit conversion

- non-finite double value
- accessing out of bounds index
- out of bounds read
- out of bounds write
- pointer subtraction
- (un)?signed downcast
- (un)?signed overflow
- function free: precondition 'freeable'
- function fclose: precondition 'valid_stream'
- function (memcpy|memmove): precondition 'valid_dest'
- function (memcpy|memmove|printf_va_1): precondition (valid_src|valid_read)
- function memcpy: precondition 'separation'
- function snprintf_va_1: precondition \valid
- function str(n)?(cpy|cat): precondition 'room_(n)?string'
- function str(n)?cat: precondition 'valid_string_dest'
- function str(n)?cpy: precondition 'valid_string_src'

3.2.2 Common Considerations

This section explains some considerations that applied to all the classes.

Frama-C halts analysis when it reaches an invalid state. These are often reported as non-terminating states. Undefined code leads to a state where anything can happen. Following that, no sound analysis makes sense, so Frama-C performs no further analysis.

Because of our misunderstanding, we unnecessarily skipped `wchar_t` cases in many classes.

Following is one subsection for each weakness class. We include details about the evaluation of a class when useful.

3.2.3 ARC/Overflow—Arithmetic Overflow

Anomalies, Observations, and Interpretations:

The good functions in 38 test cases named `CWE190_Integer_Overflow__unsigned_int_max_square_` are written wrong. Here's pertinent code from `CWE190_Integer_Overflow__unsigned_int_max_square_01.c` SARD case 235 831

```
if (abs((long)data) < (long)sqrt((double)UINT_MAX))
```

The function `labs()` should be used for the long integer instead of `abs()`. Frama-C noted the improper use of `abs()` and stopped analysis in the good function. Analysis never reached the bad function. We excused these test cases from analysis.

Results:

3628 buggy sites.

All buggy sites were found.

For ARC/Overflow, Frama-C satisfied Ockham Criterion 3.

3.2.4 ARC/Underflow—Arithmetic Underflow

Results:
2622 buggy sites.
All buggy sites were found.

For ARC/Underflow, Frama-C satisfied Ockham Criterion 3.

3.2.5 ARC/Undefined—Divide by Zero

Anomalies, Observations, and Interpretations:
Many CWE369 Juliet cases guard the division operation by comparison with 0 or with 0.000001 for floats. Frama-C's default setting does not represent a range with an "omitted middle". The option `-eva-sign-domain` handles this case with much better precision. We attributed most or all of the superfluous findings to our unrefined use of options and lumping the results of *all* runs together.

Results:
684 buggy sites.
All buggy sites were found.

For ARC/Undefined, Frama-C satisfied Ockham Criterion 3.

3.2.6 ARC/Distort—Result Distortion

Results:
1824 buggy sites.
All buggy sites were found.

For ARC/Distort, Frama-C satisfied Ockham Criterion 3.

3.2.7 ARC/Truncate—Result Truncation

Results:
684 buggy sites.
All buggy sites were found.

For ARC/Truncate, Frama-C satisfied Ockham Criterion 3.

3.2.8 ARG/Memcpy—Incorrect Argument for `memcpy()`

Results:

36 buggy sites.
All buggy sites were found.

For ARG/Memcpy, Frama-C satisfied Ockham Criterion 3.

3.2.9 BOF/Read—Read Outside Buffer

Anomalies, Observations, and Interpretations:

Some of the warnings referred to utility functions in `io.c`, where `printf()` was actually called. This would have yielded thousands of essentially repeated warnings about the utility functions. We modified the extractor to produce the location of the function calling a utility if the warning was in `io.c`.

Results:

1358 buggy sites.
All buggy sites were found.

For BOF/Read, Frama-C satisfied Ockham Criterion 3.

3.2.10 BOF/Write—Write Outside Buffer

Anomalies, Observations, and Interpretations:

The manifest has extra warnings for 74 CWE665 test cases. Here is pertinent code from `CWE665_Improper_Initialization__char_cat_17.c` SARD case 109 768:

```
char dataBuffer[100];
data = dataBuffer;

char source[100];
memset(source, 'C', 100-1); /* fill with 'C's */
source[100-1] = '\0'; /* null terminate */
strcat(data, source);
```

The buffer referenced by `data` is not initialized at all. The `strcat()` at the last line may not behave as expected. The manifest has an initialization bug for that line. The manifest also has a BOF/Write for that line, since `strcat()` could readily write outside `dataBuffer`.

Frama-C warns about the initialization problem, but doesn't bother giving any BOF/Write warning. That is perfectly reasonable behavior. For automated checking, we have the analysis change all 74 of the warnings from Frama-C for test cases named `CWE665_Improper_Initialization__char_cat_*` or `CWE665_Improper_Initialization__char_ncat_*` to BOF/Write.

The manifest has different lines for 26 `CWE122_Heap_Based_Buffer_Overflow__sizeof_struct_*` test cases. Here is pertinent code from `CWE122_Heap_Based_Buffer_Overflow__sizeof_struct_04.c` SARD case 234 207:

```
typedef struct
{
    int intOne;
    int intTwo;
} twoIntsStruct;

      twoIntsStruct * data;
37    data = (twoIntsStruct *)malloc(sizeof(data));
      if (data == NULL) {exit(-1);}
39    data->intOne = 1;
40    data->intTwo = 2;
```

The problem is that at line 37, only memory for a *pointer* to the `struct` is allocated, not memory for the whole `struct`. Frama-C determines that line 40 is undefined because the assignment is beyond the amount of memory allocated. It does not issue a warning for line 39 because it is well defined. The manifest incorrectly lists an error at line 39.

For no particular reason, instead of changing the manifest lines or having the Frama-C extractor patch the warnings, we created a file with the 26 excused warnings. The evaluation script includes this file when checking for mismatches.

Test cases under CWE242 purposely use `gets()`, even though `gets()` cannot be used safely, potentially causing BOF/Write. André Maroneze informed us that the library provided with Frama-C is missing proper preconditions for `gets()`. Since the library we used for Frama-C does not correctly model `gets()`, we excluded all CWE242 from the analysis.

Frama-C didn't give BOF/Write warnings in its default mode for the test cases using wide-character functions. André Maroneze suggested we include `wchar.c` from the Frama-C library. With that, Frama-C gave BOF/Write warnings on wide-character test cases.

Results:
5156 buggy sites.
All buggy sites were found.

For BOF/Write, Frama-C satisfied Ockham Criterion 3.

3.2.11 DEP—Dereference Erroneous Pointer

In several cases Frama-C produces the same, perfectly useful warning for classes of bugs that we distinguish. Warnings that we code as different classes are `out of bounds read` and `accessing out of bounds index`. For example, here is the pertinent code from `CWE476_NULL_Pointer_Dereference__binary_if_03.c` SARD case 104 534:

```
    twoIntsStruct *twoIntsStructPointer = NULL;
```

```
if ((twoIntsStructPointer != NULL) &
                        (twoIntsStructPointer->intOne == 5))
{
```

The use of `&` in the condition causes both statements to be evaluated. Frama-C reports `out of bounds read. assert \valid_read(&twoIntsStructPointer->intOne);`

Here is pertinent code from `CWE680_Integer_Overflow_to_Buffer_Overflow__malloc_fgets_03.c` SARD case 240 892:

```
intPointer = (int*)malloc(data * sizeof(int));
if (intPointer == NULL) {exit(-1);}
for (i = 0; i < (size_t)data; i++)
{
    intPointer[i] = 0;
}
```

To quote the comment in the code, "if data * sizeof(int) > SIZE_MAX, [the multiplication] overflows to a small value [not allocating enough memory] so that the for loop doing the initialization causes a buffer overflow". Frama-C reports `out of bounds read. assert \valid_ read(intPointer + 0);`

For analysis, we separate these into two different classes: DEP (NULL pointer dereference, in this case) and BOF/Write. In these and other situations, we used the names of the test cases to distinguish them and encode them into our comparison classes.

Anomalies, Observations, and Interpretations:

In 114 cases under CWE690, the manifest lists two flaws. Here is pertinent code from `CWE690_NULL_Deref_From_Return__struct_calloc_01.c` SARD case 111 602:

```
twoIntsStruct * data;
data = (twoIntsStruct *)calloc(1, sizeof(twoIntsStruct));
data[0].intOne = 1;
data[0].intTwo = 1;
```

The manifest has DEP flaws for both the assignment to `intOne` and the assignment to `intTwo`. As explained in 3.2.2, Frama-C warns about the first assignment when data is NULL, which is undefined behavior, then stops further analysis.

We accommodated this in automated analysis by creating a file of excused warnings with the second 114 assignments.

Results:

1204 buggy sites.

All buggy sites were found.

For DEP, Frama-C satisfied Ockham Criterion 3.

3.2.12 DEP/ICP—Incorrect Pointer Arithmetic

Anomalies, Observations, and Interpretations:

Frama-C didn't warn about cases of implementation-dependent code under CWE188. For example, here is the pertinent code from `CWE188_Reliance_on_Data_Memory_Layout__union_06.c` SARD case 82 097:

```
union
{
    struct
    {
        char charFirst, charSecond, charThird, charFourth;
    } structChars;
    long longNumber;
} unionStructLong;
unionStructLong.longNumber = 0x10203040;
/* FLAW: this operation depends on the byte-order, size,
 * alignment/packing of struct and union fields */
unionStructLong.structChars.charFourth |= 0x80;
```

André Maroneze informed us, "The memory representation of C types chosen by the compiler, namely size and alignment constraints, but also endianness, is one of the hypotheses used by Frama-C analyses".

The operation of this code is not undefined; it depends on the implementation and hardware. It is reasonable to want automated analysis to point out such implementation-dependent code, but Frama-C does not fulfill this need currently. Instead of trying different options, we skipped analysis of CWE188 test cases due to time constraints.

Results:

34 buggy sites.
All buggy sites were found.

For DEP/ICP, Frama-C satisfied Ockham Criterion 3.

3.2.13 PAR—Pointer Arithmetic

Results:

36 buggy sites.
All buggy sites were found.

For PAR, Frama-C satisfied Ockham Criterion 3.

3.2.14 INI—Initialization Fault

Anomalies, Observations, and Interpretations:

In 96 cases under CWE457, the manifest lists two flaws. Here is the pertinent code from `CWE457_Use_of_Uninitialized_Variable__struct_01.c` SARD case 103 889:

```
twoIntsStruct data;
printIntLine(data.intOne);
printIntLine(data.intTwo);
```

The manifest has INI flaws for uses of both `intOne` and `intTwo`. After warning about the use of `intOne`, Frama-C enters a "non-terminating state" and stops further analysis, as explained in 3.2.2.

We accommodated this in automated analysis by creating a file of excused warnings with the second 96 uses.

Results:

776 buggy sites.
All buggy sites were found.

For INI, Frama-C satisfied Ockham Criterion 3.

3.2.15 MAL—Memory Deallocation

Results:

784 buggy sites.
All buggy sites were found.

For MAL, Frama-C satisfied Ockham Criterion 3.

3.2.16 Summary of Evaluation

Warnings from Frama-C led us to discover 38 test cases with incorrect "good" code in CWE190 (Sec. 3.2.3).

Warnings are the union of four sets of runs, each with different options (Sec. 3.2.1).

Frama-C terminates analysis when it detects certain failures. Therefore there are no sites for Ockham purposes after a terminating failure (Sec. 3.2.2).

Frama-C always warns about buggy sites, but may warn about sites without bugs.

In many instances there are minor differences between the location of flaws given in the manifest and locations reported by Frama-C (Sec. 3.2.10).

Frama-C lacks a sufficiently detailed model for the function `gets()`. The function `gets()` is inherently dangerous and should not be used anyway (Sec. 3.2.10).

In the test cases of the 24 sets used from the Juliet 1.3 C test suite, we considered 18 826 buggy sites for Frama-C. Table 3 gives the number of buggy sites considered for each class. We processed a total of 42 056 Frama-C warnings.

Class	Bugs
ARC_Distort	1824
ARC_Oflow	3628
ARC_Trunc	684
ARC_Uflow	2622
ARC_Undef	684
ARG_Memcpy	36
BOF_Read	1358
BOF_Write	5156
DEP	1204
DEP_ICP	34
INI	776
MAL	784
PAR	36

Table 3. Number of Buggy Sites Considered for Frama-C/Eva for Each Weakness Class

Frama-C with Eva satisfied the SATE VI Ockham Sound Analysis Criteria.

4. Observations and Conclusions

4.1 New Errors Found in Juliet 1.3 and its Manifest

While evaluating Frama-C, Astrée, and another tool, we found several previously-unknown systematic problems in Juliet 1.3 and thousands of problems in its manifest of known errors. We previously evaluated Frama-C "Neon" during SATE V Ockham, which led us to find and correct many errors in the former version, 1.2, of Juliet, see Ref. [9, Sec. 3.6]. This section summarizes what we found during SATE VI Ockham.

We discovered poor code in "good" functions in CWE191 (Sec. 3.1.4) and incorrect code in "good" functions in CWE190 (Sec. 3.2.3).

Evaluating Astrée found problems in testcasesupport/io.c. The following code is at lines 116 and 117, then again at 138 and 139:

```
swscanf(&hex[2 * numWritten], L"%02x", &byte);
bytes[numWritten] = (unsigned char) byte;
```

If `swscanf()` fails, then `byte` is not initialized. At many locations in io.c, for example line 15, the possible error from `printf()` is not captured or even explicitly ignored with a cast to `void`. This is common with C programmers.

Evaluating the beta version of another tool found that all `CWE124_Buffer_Underwrite__` and `CWE127_Buffer_Underread__char_(alloca|declare)_loop_` cases have undefined behavior. The bad cases have the following line:

```
data = dataBuffer - 8;
```

making `data` point before the beginning of the buffer. This is undefined. The C11 standard [4], Sec. 6.5.6 Additive operators, paragraph 8 says, talking about the result of subtracting integers from pointers:

> If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow; otherwise, the behavior is undefined.

Evaluating another tool also found that all CWE253 cases involving `fgets()` have undefined behavior. For instance the following pertinent code is at line 34 of `CWE253_Incorrect_Check_of_Function_Return_Value__char_fgets_01.c` SARD case 92 718:

```
if (fgets(data, 100, stdin) < 0)
```

The C11 standard [4], Sec. 6.5.8 Relational operators says "both operands [must be] pointers". We believe the omission of `NULL` is not an oversight since Sec. 6.5.9 Equality operators refers to "null pointer constant" explicitly.

The manifest had many problems: missing 19 BOF/Write errors in CWE122 and locations of additional BOF/Write errors in CWE122 are incorrect (Sec. 3.1.2 and Sec. 3.2.10), missing 576 ARC/Overflow errors in CWE680 (Sec. 3.1.3), extraneous ARC/Distort errors in CWE680 (Sec. 3.1.4), spurious ARC/Truncate errors in CWE197 (Sec. 3.1.7), missing ARC/Truncate errors in CWE369 (Sec. 3.1.7), missing thousands of BOF/Read errors (Sec. 3.1.9), "fossil" BOF/Read errors in CWE121 and CWE122 (Sec. 3.1.9), missing a total of 552 BOF/Write errors in CWE121, CWE122, and CWE124 (Sec. 3.1.10), incorrect location of INI errors and also missing BOF/Write errors in CWE665 (Sec. 3.1.10), and incorrect location of MAL/Double Free errors in CWE415 and CWE761 (Sec. 3.1.16).

4.2 Weakness Classes

Although the SATE VI Ockham Sound Analysis Criteria used the term "weakness classes," no classes are specified. For evaluation we defined our own classes, see Sec. 2.4. We tried to be clear and logical in our choice of classes, but they still did not always correspond well to the warnings that the tools used.

It may have been easier to evaluate the warnings as produced by the tools. Several times, we communicated with the tool developers to better understand exactly what the tool was reporting.

Without understanding the exact definition of the class of weakness the tool was considering, we could not decide whether a known bug corresponded to a tool warning: perhaps there was just a difference in choice of which line number to report. If the tool was intended to report that class, then a missing warning indicated an error. If the tool in actuality is not

considering a particular class of warning, such as integer overflow of types smaller than `int`, then a known bug should be ignored.

In retrospect, there is little need to have DEP/Incorrectly Computed faults as a separate class. It has less than 5% of all DEP cases.

4.3 Summary

We processed a total of 78 372 warnings over 29 sets from the Juliet 1.3 C test suite.

Both Astrée and Frama-C with Eva satisfied the SATE VI Ockham Sound Analysis Criteria.

Acknowledgments

We thank AbsInt Angewandte Informatik GmbH for an evaluation license to run Astrée and Dr.-Ing. Jörg Herter, Christoph Mallon, and Dominik Erb for answering our many questions about it. We thank the List Institute, Commissariat à l'énergie atomique et aux énergies alternatives (CEA) for making Frama-C and Eva available and André Maroneze, Florent Kirchner, and David Bühler for answering our many questions about it.

References

[1] (2016) Software Assurance Metrics And Tool Evaluation. Available at https://samate.nist.gov/.

[2] Delaitre A, Stivalet B, Black PE, Okun V, Ribeiro A, Cohen TS (2018) SATE V report: Ten years of static analysis tool expositions (National Institute of Standards and Technology), SP 500-326. https://doi.org/10.6028/NIST.SP.500-326

[3] (2014) Static Analysis Tool Exposition (SATE) V. Available at https://samate.nist.gov/SATE5.html.

[4] (2011) ISO/IEC 9899:2011 programming languages - C, Committee Draft — April 12, 2011 N1570 (The International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC) Joint Technical Committee JTC 1, Information technology, Subcommittee SC 22, Programming languages, their environments and system software interfaces, Working Group WG 14 - C), Available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf.

[5] (2019) Fast and sound runtime error analysis. Accessed 26 August 2019. Available at https://www.absint.com/astree/.

[6] (2019) What is Frama-C. Available at http://frama-c.com/what_is.html.

[7] AdaCore and Thales (2018) Implementation guidance for the adoption of SPARK. Available at https://www.adacore.com/uploads/books/pdf/ePDF-ImplementationGuidanceSPARK.pdf.

[8] AdaCore, Ltd AU (2019) SPARK 2014 user's guide. Accessed 21 February 2020. Available at http://docs.adacore.com/spark2014-docs/html/ug/.

[9] Black PE, Ribeiro A (2017) SATE V Ockham sound analysis criteria (National Institute of Standards and Technology), IR 8113. https://doi.org/10.6028/NIST.IR.8113
[10] (2016) Software Assurance Reference Dataset (SARD). Available at https://samate.nist.gov/SARD/.
[11] Black PE (2018) Juliet 1.3 test suite: Changes from 1.2 (National Institute of Standards and Technology), TN 1995. https://doi.org/10.6028/NIST.TN.1995
[12] Stoneburner G, Hayden C, Feringa A (2004) Engineering principles for information technology security (a baseline for achieving security), revision a (National Institute of Standards and Technology), SP 800-27 Rev. A. https://doi.org/10.6028/NIST.SP.800-27rA. Withdrawn on 15 November 2017. Superseded by NIST SP 800-160.
[13] Common weakness enumeration. Accessed 23 August 2019. Available at https://cwe.mitre.org/.
[14] Bojanova I, Black PE, Yesha Y, Wu Y (2016) The bugs framework (BF): A structured approach to express bugs. *2016 IEEE International Conference on Software Quality, Reliability, and Security (QRS)*, pp 175–182. https://doi.org/10.1109/QRS.2016.29. Vienna, Austria
[15] (2019) The Bugs Framework (BF). Available at https://samate.nist.gov/BF/.
[16] RTCA (2011) *Formal Methods Supplement to DO-178C and DO-178A*. DO-333.
[17] (2015) Rice's theorem. Available at http://en.wikipedia.org/wiki/Rice's_theorem.
[18] (2018) XZ Utils. Available at http://tukaani.org/xz/.
[19] (2018) Open(2). Accessed 30 August 2019. Available at http://man7.org/linux/man-pages/man2/open.2.html.
[20] (2019) Proving formal properties for critical software. Available at http://frama-c.com/features.html.