NISTIR 8113

SATE V Ockham Sound Analysis Criteria

Paul E. Black
Athos Ribeiro

This publication is available free of charge from:
http://dx.doi.org/10.6028/NIST.IR.8113

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

NISTIR 8113

SATE V Ockham Sound Analysis Criteria

Paul E. Black
Software and Systems Division
Information Technology Laboratory

Athos Ribeiro
Department of Computer Science
University of São Paulo

This publication is available free of charge from:
http://dx.doi.org/10.6028/NIST.IR.8113

March 2016

Abstract

Static analyzers examine the source or executable code of programs to find problems. Many static analyzers use heuristics or approximations to handle programs up to millions of lines of code. We established the Ockham Sound Analysis Criteria to recognize static analyzers whose findings are always correct. 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.

In Static Analysis Tool Exposition (SATE) V, only one tool was submitted to be reviewed. Pascal Cuoq and Florent Kirchner ran the August 2013 development version of Frama-C on pertinent parts of the Juliet 1.2 test suite. We divided the warnings into eight classes, including improper buffer access, NULL pointer dereference, integer overflow, and use of uninitialized variable. This document details the many technical and theoretical challenges we addressed to classify and review the warnings against the Criteria. It also describes anomalies, our observations, and interpretations. Frama-C reports led us to discover three unintentional, systematic flaws in the Juliet test suite involving 416 test cases. Our conclusion is that Frama-C satisfied the SATE V Ockham Sound Analysis Criteria.

1. Background

1.1. SATE

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 a set of programs as test cases. Participating tool developers run their tool on the test cases and submit their results (tool reports). The organizers 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,
- Speed adoption of tools by objectively demonstrating their use on real software.

In SATE V [2], the SAMATE team introduced the SATE V Ockham Sound Analysis Criteria, a track for static analyzers whose findings are always correct. Tools do not have to be "bug-finders," that is, report flaws or bugs. The Ockham Criteria also applies to tools that report that sites are certainly bug-free.

Section 2 explains the Criteria in detail. It also presents the general procedure to evaluate a tool against the criteria. Section 3 explains how the procedure is instantiated for the only participant in SATE V Ockham, Frama-C, and details of the evaluation. Section 4 lists ideas to improve future Ockham Criteria exercises. Our conclusions are in Sec. 5.

2. The Criteria

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 criteria name always includes a time reference: SATE V Ockham Sound Analysis Criteria.

The value of a sound analyzer is that every one of its findings can be assumed to be correct, even if it cannot handle enormous pieces of software or does not handle dozens of weakness classes. In brief the criteria are:

1) There is a claim that the tool's findings are always correct.
2) The tool produces findings for most of the program.
3) Even one incorrect finding disqualifies a tool.

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

We use the term *warning* to mean a single report produced by a tool. For instance, `integer overflow at line 14` is a warning. A *finding* may be a warning or it may be a site with no warning. For instance, a tool may be implemented to be cautious 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.

A tool might be sure of bugs at some sites and sure of absence of bugs at other sites. However it might not be sure about some sites. For such a tool, a finding is only those things that it is sure about, both buggy sites and sites without bugs.

2.1. Details

This section has the details of the Criteria. First we give the three formal criteria, then we follow with definitions, informative statements, and discussion.

We tried to set requirements that communicated our intent, ruled out trivial satisfaction, and were understandable. We announced that we planned to be liberal in interpreting the rules: we hoped tools would satisfy the criteria, so we gave tools the benefit of a doubt.

The formal 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 60 % of buggy sites OR of non-buggy sites.
3) Even one incorrect finding disqualifies a tool for this SATE.

No manual editing of the tool output was allowed. No automated filtering specialized to a test case or to SATE V was allowed, either.

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

We used the term *sound* to mean that every finding was correct. The tool need not produce a finding for every site; that is completeness. See Sec. 2.3 for a discussion of our use of the terms "sound" and "complete."

A tool may have settings that allow unsound analysis. The tool still qualified if it had clearly sound settings. *For example, for fast analysis or for some classes of weaknesses, the user may be able to select unsound (approximate) function signatures.* A more inclusive statement of Criterion 1 is, the tool is claimed to be sound or has a mode in which analysis is sound.

Criterion 2 deals with the number of findings produced: The tool produces findings for a minimum of 60 % of sites.

This criterion was included since it is impossible in theory for an algorithm to correctly report the presence of a property in all software (and not report it when the property is absent). [3] Rice's Theorem states that either an algorithm fails to report the property in some cases when the property is present, or it may incorrectly report the property's presence when it is absent.

We chose the former: the Ockham Criteria emphasized the approach in which findings need no further examination of their validity. Rice's Theorem then says that we must accept that in some cases a sound tool may not be able to decide whether or not a site has a weakness.

Without this criterion, a trivial tool could produce no findings, which is arguable not incorrect, and satisfy the Ockham Criteria. For a tool to be useful, it must produce findings for many sites in many pieces of software.

After consultation with the SATE program committee, we chose 60 % as a level that would be useful, yet readily achievable by current tools. In the future, we will likely set a higher limit.

A *site* is a location in code where a weakness might occur. *For instance, every buffer access in a C program is a site where buffer overflow might occur if the code is buggy. In other words, sites for a weakness are places that must be checked for that weakness.* See Sec. 2.2 for further exposition of what constitutes a site.

A buggy site is one that has an instance of the weakness. That is, there is some input that will cause a violation. A non-buggy site is one that does not have an instance of the weakness, in other words, is safe or not vulnerable.

Data flow weaknesses, such as SQL injection, have a notion of connected source/sink pairs. A site is closely related to the notion of a sink. *A program may accept input at several places, but access SQL in just one or two places. Other programs may have just a few inputs from which data flows to many SQL calls. Counting pairs may balance these styles.*

A tool is allowed to have a different definition of site, as long as it is expressed.

A *finding* is a definitive report about a site. In other words, that the site has a specific weakness (is buggy) or that the site does not have a specific weakness (is not buggy). *Tentative 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.*

A tool produces *warnings* about sites. A finding may be a warning or it may be the *lack* of a warning. *If the tool uses conservative approximations, it may produce false alarms, that is, warnings about sites that are actually not buggy. However, for such a tool, the* lack *of a warning for a site is a finding that the site is definitely safe.*

We chose to use test cases from SATE V. SATE V offered large, production programs and the Juliet 1.2 test suite. [4] Juliet cases are small, synthetic programs. Both production programs and Juliet have cases in C or in Java.

To be useful, a tool must be able to handle one of the large programs or many of the Juliet test cases. All the Juliet test cases in the appropriate language and weakness class(es) are considered one test case. *For instance, a tool cannot achieve the criteria for running just a selection of buffer overflow cases; it must run on all Common Weakness Enumeration (CWE) 121, 122, 123, 124, 126, 127, 129, and 131 cases.*

Processing different classes of weaknesses may take very different software 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.

Related to the minimum number of sites criterion above, to be significant, there had to be a minimum of 10 sites and two findings in the test case. *Reporting no possible buffer overflows for a Java program or no uncaught exceptions for a C program is not grounds for satisfying SATE V Ockham Criteria.*

We hoped that tools would be run on production test cases. To prepare for that, we considered how we might check a tool's results. We decided to estimate the number of sites in a test case by simple programs if there was significant disagreement or uncertainty about the number. *We anticipated accepting tools' count of number of sites. If there were concerns, we would ad hoc "grep" or similar, simple methods.*

We would determine that findings were correct (or incorrect) by simple programs for Juliet test cases. For other test cases, we anticipated comparing tool findings by simple programs and manually reviewing differences. We had planned to compare Ockham results with SATE V results for additional confidence.

What if there were unexpected findings? All reasoning is based on models, assumptions, definitions, etc. (collectively, "models"). Unexpected findings resulting from model differences need not disqualify a tool. In consultation with the tool maker, we would decide if an unexpected finding results from a reasonable model difference or whether it is incorrect. To satisfy the SATE V Ockham Criteria, any such differences must be publicly reported.

For instance, one tool may assume that file system permissions are set as the test case requires, while another tool makes no assumptions about permissions (that is, assumes the worst case). In this case, the tools could have different findings, yet both are correct.

However, if a tool modeled the "+" operator as subtraction, it was incorrect.

We realized that models are hard to build and validate, but the value of a sound analyzer is its correctness. We had thought that a proof of correctness of analysis soundness at the mathematical specification level might excuse incorrect findings.

We planned to have the Ockham Criteria many times. If a tool did not satisfy the SATE V Ockham Criteria, it might satisfy the Ockham Criteria in the future. Hence, Criterion 3 stated that even one incorrect finding disqualified a tool for this SATE, meaning SATE V.

2.2. Definition of "site"

As stated above, a *site* is a location in code where a weakness might occur. For instance, every buffer access in a C program is a site where buffer overflow might occur if the code is buggy. In other words, sites are places that must be checked. The determination of a site depends on local information. That is, global or flow-sensitive information should not be needed to determine where sites are in code.

For example, the following code comes from Software Assurance Reference Dataset (SARD) [5] case 62 804. It has one site of writing to a buffer, `data[i] =`, which needs to be checked for a write-outside-buffer bug. There is also one site of reading from a buffer, `source[i]`, where the program might read outside the buffer if there is a bug.

```
for (i = 0; i < 10; i++)
{
    data[i] = source[i];
}
```

In addition, the code has sites of uninitialized variable, every place that `i` is used, and an integer overflow site, `i++`. Notice that the assignment statement in the body of the loop has several sites: a write buffer site, a read buffer site, and sites where variables are used.

The concept of "site" is similar to the concept of "foothold" used by Nikolai Mansourov in the description of Software Fault Patterns [6].

One exception to locality is dead code. For some purposes, code that is unreachable or can never be executed should not be considered. For other purposes, all code should be considered, since a piece of code's possible execution might easily change with a minor alteration to the source.

Locations in code are often excluded as sites because of local information. For example, consider the weakness class CWE-369: Divide By Zero. A simple definition of site for this class is just every occurrence of a division operator (`/`). Consider the division operator in the code fragment `mid = height/2`, which is division by a constant. Since division by a constant other than zero is clearly never a divide by zero and this situation can be detected easily, we may exclude division by a non-zero constant as a site for divide by zero.

Another way to think of it is that a site is the last place in code that the programmer may make necessary checks or modify state. For instance, the C standard library function `strcpy()` receives two pointers to locations, presumably arrays. It does not get enough information to determine if the destination array is big enough to hold the source string. Thus after a call to `strcpy()`, the programmer can no longer influence whether or not a violation occurs; necessary information is not available. Therefore the call to `strcpy()` is considered the site of a buffer overflow even though the violation occurs in the library routine. Similarly when the programmer's code invokes a primitive operator, such as `[ ]`. The data state needs to satisfy the preconditions of the library function or operator.

2.3. About "Sound" and "Complete" Analysis

The term *sound* and the term *complete* are used differently by different communities. In this section, we explain that two different pairs of meanings both have valid reasons.

Most of the theorem proving, formal methods, and static analysis communities use "sound" to mean that all bugs are reported and "complete" to mean that every bug report is a correct report. We explain this usage below. For the Ockham Criteria, we used "sound" to mean that every finding[1] was correct. We used "complete" to convey the meaning of a finding for every site.

2.3.1. Ockham Criteria Use of the Term Sound. The terms "sound" and "complete" come from logic. A deductive system consists of axioms and inference rules. Axioms and statements are expressed in a specific language. A deductive system is *sound* if and only if every statement that can be deduced is true. On the other hand, if every true statement can be deduced, the deductive system is *complete* [7].

The differences in use between us arise in applying those terms to software.

For the purposes of the Ockham Criteria, logical statements corresponded to declarations of properties at a location in software or properties in a specific piece of software. For example, here are two statements:

- For all executions that reach location q, the variable `x` has the value 7.
- There exists an input value such that the access to buffer `buf` at location r writes outside the buffer. (We usually express this as, `buf` has a buffer overflow at r.)

The software is understood to be included in the statement of a property. For instance, a complete statement of the first property is: "For all executions of software S that reach location q, the variable `x` has the value 7."

Axioms and rules of inference include the semantics of the software's language.[2] Axioms and inference rules are written into static analyzers. In addition, some static analyzers have a general language of properties. That is, users can write rules, and the static analyzer's engine checks software against those rules. Some static analyzers have a fixed set of properties, which are embedded in the software as well.

With these meanings, a sound deductive system corresponds to a static analyzer that only reports (derives or proves) properties that are true. A complete deductive system corresponds to a static analyzer that reports, or could report, all true properties that are within the analyzer's domain.

2.3.2. Other Meaning of "Sound" and "Complete". The other meaning of "sound" and "complete" is best understood by considering how static analyzers employ abstract interpretation. For this purpose, abstract interpretation deduces properties of software by symbolically executing (interpreting) the software. For efficiency, the software is only interpreted once (or just a few times). The challenge for analysis is to maintain all the possible states that the software might have during the single interpretive run. Unfortunately no analysis program can efficiently embody the exact set of values even one variable might have in all cases, let alone all states. As an extreme instance, a variable may be a large prime number in cryptographic software. Another example is that the value of a variable may indicate the form or content of a data structure. Any abstract interpretation will only handle a few ways of embodying possible values. Some ways are as a small set of discrete values, as a range of possible values, or as a linear relation between the values of two variables.

Abstract interpretation approximates concrete or actual states of software by lists of embodiments of

1. A "bug report" or warning is not necessarily the same thing as a "finding." A finding may be a bug report. In addition or instead, a finding may be a site that does *not* have a bug report, that is, the tool is sure is not buggy.

2. One should be clear as to whether the semantics correspond to a language standard, the behavior of code produced by a particular compiler, or something else.

values of variables. These approximations are a part of abstract states that represent concrete states of the software being analyzed.

In deterministic software, one concrete state transitions to another concrete state. Similarly, each abstract state makes an abstract transition to another abstract state.

The other meaning of sound analysis has two motivations, both of which lead to the same effect. The first motivation is that analysis never misses any weakness or vulnerability. The second motivation is that the abstract states in a trace represent *all* the concrete states that may be reached. More specifically, if a (start) concrete state transitions to an (end) concrete state, the abstract state that represents the start concrete state makes a corresponding abstract transition to an abstract state that represents the end concrete state.

For either motivation, the effect is that abstract states and transitions must overapproximate concrete states and transitions, given the practical limits of embodying variable values. The effect of overapproximation is that abstract interpretation often leads to abstract states that include vulnerable concrete states, even though the vulnerable concrete states never can occur.

For instance, consider checking for a divide-by-zero failure in the following code fragment:

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

Suppose that analysis only embodies possible values of a variable in two ways: as a single value or as a range of values from a minimum to a maximum. After the first line, `x` can have any `int` value. This can be embodied exactly as a range from the minimum `int` to the maximum `int`. Immediately after the conditional, the possible values of `x`, all values omitting zero, cannot be embodied exactly. The only practical solution is to continue to embody the possible values as the entire range. When analysis checks the next line, zero is found to be a possible value. Analysis reports a (possible) divide-by-zero, even though it cannot actually occur.

Of course, for any particular situation, an embodiment can be devised to handle it correctly. However the situations occurring in actual pieces of software are essentially limitless.

Hence sound analysis in this second sense will typically produce false alarms (false positives), but never miss a possible problem (no false negatives).

By analogous argument, complete analysis never has false alarms, but may miss some problems.

Although related, the two pairs of meanings are exactly reversed only in a specific situation of the Ockham Criteria: when findings are the tool's report of bugs and sites are the buggy sites. In this situation, sound in the Ockham Criteria means all tool reports are of buggy sites, that is, no false alarms. Ockham Criteria complete means that every buggy site is reported by the tool, that is, no false negatives.

2.4. General procedures

This section describes the general procedure that we established to confirm that a participating tool satisfied the Criteria.

1) Decide what constitutes a site.
2) Determine the list of sites
$$U = the\ set\ of\ all\ sites$$
3) Determine the list of findings
$$F = the\ set\ of\ all\ findings$$
Recall that findings may be buggy sites or good (non-buggy) sites or both. Hence the use of both B and G later.
4) Check that all findings are at sites
$$F \subseteq U$$
which they should be, by definition.
   - If that is not true, reconcile the definitions of "site" and "finding."
5) Determine which sites are buggy (or non-buggy)
$$B = the\ set\ of\ all\ buggy\ (bad)\ sites$$
$$G = the\ set\ of\ all\ non\text{-}buggy\ (good)\ sites$$
   - A site is either buggy or good, but not both. By definition:
$$U = B \cup G$$
$$B \cap G = \emptyset$$
   - To achieve higher assurance, we can review SATE V reports against U, B, and G.
6) Check that
$$|F| \geq 0.6 \times |B|\ (or\ |G|\ as\ appropriate)$$
where $|F|$ is the number of items in set F, $|B|$ is the number of items in set B, etc.
   - If that is true, Criterion 2 is satisfied.
7) Check that
$$F \cap G\ (or\ B) = \emptyset$$
   - If that is true, Criterion 3 is satisfied.
   - If that is not true, it may be differences in definitions.

3. SATE V Evaluation

There was only one participant in SATE V Ockham Sound Analysis Criteria: Frama-C. Pascal Cuoq and Florent Kirchner ran the August 2013 development version. (Changes were released to the open-source engine in version 20140301 "Neon.")

3.1. Frama-C Evaluation

Frama-C is a suite of tools to analyze software written in C [8]. It is free software licensed under the GNU Lesser General Public License (LGPL) v2 license[3].

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" [8].

This satisfies Criterion 1.

3.1.1. Frama-C specific procedures. This section explains the specific adaptions of the general procedures for Frama-C.

Some situations in the C language, such as an integer overflow or left shift more than data type size have "undefined behavior," which is more drastic than "the result may be any number": no further analysis is reasonable. See Sec. A.3 for further explanation.

Frama-C issues a warning and terminates analysis when it detects that the resulting state may be undefined. Consequently sites following a terminating failure (T) have no judgments made at all, neither buggy nor non-buggy. The universe of sites is therefore syntactic sites (S) Until (**U**) a terminating failure.

$$U = S \mathbf{U} T$$

Pascal Cuoq and Florent Kirchner sent two files of warnings each from a different sets of runs of Frama-C. One set of runs modeled that every allocation failed, and the other runs modeled that every allocation succeeded. Frama-C must assume allocation failure in order to catch a possible NULL pointer dereference, for example in the following code, which comes from SARD [5] case 74 328:

```
char * dataBuffer = malloc(100);
memset(dataBuffer, 'A', 100-1);
```

Frama-C could not model both allocation failure and allocation success in one run. Warnings are the union of warnings from both files.

Frama-C always warns about a bug at a site when there is a bug, that is, there are no false negatives. Note that because of the limitations of Frama-C's models, it may warn of a bug when there is no bug, that is, there may be false alarms; see the discussion about divide by zero in Sec. 2.3.2 as one example. The reader may ask, do these false alarms disqualify Frama-C? No, because for Frama-C a *finding* is that a site is *not* buggy. If Frama-c does not produce a warning for a site, then that site is definitely not buggy. In other words, given that W is the set of all warnings,

$$F = U - W$$

By definition, the consistency check in step 4, $F \subseteq U$, was trivially satisfied. However, we gained confidence by checking that all warnings are sites. Therefore we substituted the consistency check

4) Check that
$$W \subseteq U$$
If that was not true, reconcile definition of site and finding.

To determine buggy sites, we developed a "master list" from the comments and repeated structures in Juliet code. When we found inconsistencies, we investigated and resolved them as needed. We improved the code to scan Juliet for sites and determine which one were buggy, the Juliet master list, and other converters and extractors. Since findings were good sites for Frama-C, the criteria checks were

6) Check that
$$|F| \geq 0.6 \times |G|$$
7) Check that
$$F \cap B = \emptyset$$
If that was not true, investigate the reason including definition of site and assignment of warning.

Since $G = U - B$ (and $B \subseteq U$)[4], we can rewrite step 6 so we only used buggy (B) sites:

6) Check that
$$|F| \geq 0.6 \times (|U| - |B|)$$

3.2. Implementation

We performed the bulk of the analysis with automated scripts and custom programs. The general flow was to (1) extract appropriate sites from the Juliet tests, (2) extract and interpret appropriate warnings from the Frama-C report, and (3) match and process the two extracts in various ways.

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

4. We need to know that $B \subseteq U$ because in general,
$$|U - B| = |U| - |B| + |B - U|$$
Since $B \subseteq U$, $|B - U| = 0$, therefore $|U - B| = |U| - |B|$.

Automated scripts allowed us to rerun with relative ease when we needed to.

Some exclusions and special handling were built into the code. These are mentioned where we discuss the exclusions or special handling, such as Sections 3.4.1, 3.4.5, and 3.4.7.

All the scripts and files are available in a tar file with xz compression [9] at DOI `http://dx.doi.org/10.18434/T4WC7V` or `https://s3.amazonaws.com/nist-ockham-criteria-sate-v-data/ockhamCriteriaSATEVdata.tar.xz`
The README is available at `https://s3.amazonaws.com/nist-ockham-criteria-sate-v-data/README`

3.3. Common considerations

We divided the Frama-C warnings into classes and examined them generally class by class. This section explains some considerations that applied to all the classes.

3.3.1. Analysis termination after `RAND32()` macro. The Juliet 1.2 test suite uses a macro, `RAND32()`, defined as follows:

```
#define RAND32() \
  ((rand()<<30) ^ (rand()<<15) ^ rand())
```

The International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC) C 2011 standard Sec. 6.5.7 Bitwise shift operators says, "If the value of the right operand is negative or is greater than or equal to the width in bits of the promoted left operand, the behavior is undefined." [10]

Frama-C models `rand()` as returning a type that is less than 30 bits. According to the standard, the result of executing a statement with `RAND32()` is undefined. As explained in Sec. 3.1.1, Frama-C does no further analysis.

Our site extraction is largely syntactic or local, so it was difficult to exclude sites that followed undefined behavior. Given the limitation of our analysis, we completely excluded the 76 test cases that use `RAND32()`. See Sec. A.3 for details.

3.3.2. Cases Under CWE191 Not Processed. During evaluation, we observed that there were no warnings at all for test cases under CWE191. Upon inquiry, we learned that because of a simple human mistake, Frama-C was not run on any cases under CWE191. See Sec. A.1 in the appendix for more detail.

We decided to exclude all sites under the CWE191 subdirectory from the analysis to avoid misinterpretations in the final results.

The developers later submitted files with the warnings. We decided not to evaluate those since they were obtained with a later version of Frama-C.

3.4. Evaluation by weakness classes

We sent a set of Juliet 1.2 test cases containing the following CWEs to those running Frama-C:

- CWE-121 Stack-based Buffer Overflow
- CWE-122 Heap-based Buffer Overflow
- CWE-123 Write-what-where Condition
- CWE-124 Buffer Underwrite
- CWE-126 Buffer Over-read
- CWE-127 Buffer Under-read
- CWE-190 Integer Overflow
- CWE-191 Integer Underflow
- CWE-369 Divide by Zero
- CWE-457 Use of Uninitialized Variable
- CWE-476 NULL Pointer Dereference
- CWE-562 Return of Stack Variable Address

The result we received from them had the following nine warnings:

- division by zero
- floating-point NaN or infinity
- invalid arguments to library function
- invalid memory access
- making use of address of object past its lifetime
- overflow in conversion
- passing INT_MIN to standard function abs()
- reading from uninitialized lvalue
- undefined arithmetic overflow

The warnings did not match simply to CWE classes. We came up with nine classes of weaknesses. By examining verbose information that Frama-C supplied with each warning, we matched most warnings to one of the weakness classes. Some warnings did not fit into those classes or were not readily handled by our automatic processing. We explain those in Sec. 3.5.

Following is one subsection for each weakness class with some details about the evaluation of the class.

3.4.1. Write Outside Buffer. This includes CWE-121 Stack-based Buffer Overflow, CWE-122 Heap-based Buffer Overflow, and CWE-124 Buffer Underwrite ('Buffer Underflow'). Frama-C does not distinguish between stack-based and heap-based buffers. For the Ockham Criteria, the distinction between stack-based and heap-based or between underflow and overflow is not important. Either the buffer is always accessed properly or there is a bug, which should be fixed or mitigated.

Site definition:

- Write to an array (buffer), either by `[]` or unary `*`. Specifically array access on the left hand side of an assignment or use as a destination in a standard library function. The exception is that `memcpy()` or `memmove()` into a structure is not a site.

Programmers often use `memcpy()` and `memmove()` to fill or move entire structures. The Frama-C model allows them to copy or move anything to anywhere.

Anomalies, Observations, and Interpretations:

The version of Frama-C that was used for the Ockham Criteria, the August 2013 development version, did not support wide string literals, e.g. `L"Good"`, nor the format specifier for wide string (`%ls`). For this reason, we did not include sites with wide string literals or the wide string format specifier in the Universe. Neither did we include sites with wide character arrays that are passed to `printWLine()`, which uses the wide string format specifier. These exclusions are coded in `extractor.py`. For more details, see Sec. A.2 in the appendix.

Results:

97 678 sites ($|U|$). 18 767 warnings ($|W|$). 78 911 findings ($|F|$). 7400 buggy sites ($|B|$).

We confirmed the following.

Warnings and sites were consistent: $W \subseteq U$

There were many findings: $|F| \geq 0.6 \times (|U| - |B|)$

All findings were correct: $F \cap B = \emptyset$

For Write Outside Buffer, which includes CWE-121, CWE-122 and CWE-124, Frama-C satisfied the criteria.

3.4.2. CWE-123 Write-what-where Condition.

Site definition:

- Use of *, `->`, or `[]` operators.

A more generally complete definition of site for this weakness should include calls to some library functions. However the above definition was sufficient for Juliet test cases.

Results:

72 084 sites ($|U|$). 791 warnings ($|W|$). 71 293 findings ($|F|$). 228 buggy sites ($|B|$).

We confirmed the following.

Warnings and sites were consistent: $W \subseteq U$

There were many findings: $|F| \geq 0.6 \times (|U| - |B|)$

All findings were correct: $F \cap B = \emptyset$

For CWE-123 Write-What-Where, Frama-C satisfied the criteria.

3.4.3. Read Outside Buffer.

This includes CWE-126 Buffer Over-read and CWE-127 Buffer Under-read. Frama-C did not distinguish between read before the beginning of buffer and read after the end of buffer. For the Ockham criteria, the difference is not important.

Site definition:

- Read from an array (buffer), either by `[]` or unary `*`.

The access could be in an expression or it could be embedded in the left hand side of an assignment. For example `a[b[i]] = ...` reads buffer `b`.

Anomalies, Observations, and Interpretations: Some warnings deal with invalid argument to `printf()`: `invalid arguments to library function for printf`. We deemed these to be Read Outside Buffer warnings since they would only happen to strings that are not null terminated that could lead `printf()` to an overread.

72 files in Juliet had an unintentional bug, see Sec. A.5 in the appendix. The source string passed to `memcpy()` or `memmove()` was shorter than the size to be copied or moved. In this cases, Frama-C correctly pointed out a minimal read after end of buffer for the function call. Since these flaws were unintentional, we excluded them as sites for the Ockham Criteria.

Results:

66 665 sites ($|U|$). 3396 warnings ($|W|$). 63 269 findings ($|F|$). 2168 buggy sites ($|B|$).

We confirmed the following.

Warnings and sites were consistent: $W \subseteq U$

There were many findings: $|F| \geq 0.6 \times (|U| - |B|)$

All findings were correct: $F \cap B = \emptyset$

For Read Outside Buffer, which includes CWE-126 and CWE-127, Frama-C satisfied the criteria.

3.4.4. CWE-476 NULL Pointer Dereference.

Site definition:

- Use of *, `->`, or `[]` operators.

A more generally complete definition of site for this weakness should include calls to some library functions. However the above is enough for Juliet.

Anomalies, Observations, and Interpretations: It was very difficult to distinguish the Frama-C warnings for this class from those for array access out-of-bounds. Therefore, we only included “invalid memory access” warnings for test cases in the CWE476 subdirectory.

Results:

72 084 sites ($|U|$). 303 warnings ($|W|$). 71 781 findings ($|F|$). 271 buggy sites ($|B|$).

We confirmed the following.

Warnings and sites were consistent: $W \subseteq U$

There were many findings: $|F| \geq 0.6 \times (|U| - |B|)$
All findings were correct: $F \cap B = \emptyset$
For CWE-476 NULL Pointer Dereference, Frama-C satisfied the criteria.

3.4.5. CWE-190 Integer Overflow or Wraparound.

Site definition:

- Use of +, ++, * (multiplication), +=, and *=. This includes array indexing (and array index scaling), hence: `[]`, too.

We did not consider divisions as sites for integer overflow. Integer division overflows **only** when dividing by zero. Since that is divide by zero, which is more specific, we excluded division as a site for integer overflow. By the same reasoning, we excluded uses of the modulo (%) operator.

Frama-C only identified signed arithmetic overflows involving types of width `int` or greater. To simplify site identification, we excluded sites from files with `_char_`, `_short_`, or `_unsigned_` in the file name. This exclusion is coded in `get_integer_inc_sites.py`. This excluded 7113 files (6105 `char`, 504 `short`, and 504 `unsigned`) in 4876 test cases (4192 `char`, 342 `short`, and 342 `unsigned`).

Anomalies, Observations, and Interpretations:
As in previous classes, we did not include bugs depending on use of the `RAND32()` macro in the findings, since it would generate mismatches during our analysis due to analysis termination, as mentioned before.

Results:
29 907 sites ($|U|$). 1356 warnings ($|W|$). 28 551 findings ($|F|$). 1026 buggy sites ($|B|$).
We confirmed the following.
Warnings and sites were consistent: $W \subseteq U$
There were many findings: $|F| \geq 0.6 \times (|U| - |B|)$
All findings were correct: $F \cap B = \emptyset$
For CWE-190 Integer Overflow, Frama-C satisfied the criteria.

3.4.6. CWE-369 Divide by Zero.

Site definition:

- Use of /, %, /=, and %=[5]. This includes all arithmetic types, including float and double computations.
- This does *not* include cases in which the right hand side is a constant, e.g. `height/2`.

Anomalies, Observations, and Interpretations:
CWE-369 Juliet cases are built with only three variants: (1) a variable, `data`, which is later used as the divisor, is set to a non-zero constant, (2) `data` is set to zero or to the return value of the function that may be 0, for instance `rand()` or `scanf()`, or (3) the division operation is guarded by comparison with 0 or with 0.000001 for floats. Frama-C's implementation of abstract interpretation cannot handle a range with an "omitted middle." This leads to a divide by zero warning even when the operation is properly guarded, as explained in Sec. 2.3.2. Most or all of the incorrect warnings, and therefore the relatively low number of findings, are attributed to this implementation choice.

Results:
3018 sites ($|U|$). 1399 warnings ($|W|$). 1619 findings ($|F|$). 684 buggy sites ($|B|$).
We confirmed the following.
Warnings and sites were consistent: $W \subseteq U$
There were many findings: $|F| \geq 0.6 \times (|U| - |B|)$
All findings were correct: $F \cap B = \emptyset$
For CWE-369 Divide by Zero, Frama-C satisfied the criteria.

3.4.7. CWE-457 Use of Uninitialized Variable.

Site definition:

- When the value of a variable is used.

In some instances after an uninitialized variable is reported, Frama-C did not produce any further warnings. We did not determine whether this is due to an undefined program state, as explained in Sec. 3.3.1, a clean-up to avoid repeated warnings about essentially the same problem, or something else.

We handled this by only including the first buggy site in a file. That is, the first buggy site is included, and any subsequent buggy site in the same file is excluded. This exclusion is coded in `get_variable_ref_sites.py`.

Results:
263 520 sites ($|U|$). 770 warnings ($|W|$). 262 750 findings ($|F|$). 560 buggy sites ($|B|$).
We confirmed the following.
Warnings and sites were consistent: $W \subseteq U$
There were many findings: $|F| \geq 0.6 \times (|U| - |B|)$
All findings were correct: $F \cap B = \emptyset$
For CWE-457 Uninitialized Variable, Frama-C satisfied the criteria.

3.4.8. CWE-562 Return of Stack Variable Address.

Site definition:

- Return statements that return an expression. This does not include return of constant values.

Anomalies, Observations, and Interpretations:
There was significant mismatch between our site definition and Frama-C's warning. Our site definition, which was based on the CWE, was in

5. Juliet includes modulo (%) operator in divide by zero.

the statement where a stack address is returned. Frama-C reports the statement where an expired address is used. Consider the following code from `CWE562_Return_of_Stack_Variable_Address__return_buf_01.c`, SARD [5] case 105 491:

```
static char *helperBad() {
    char charString[] = "helperBad string";
    return charString;
}

{
    . . .
    printLine(helperBad());
    . . .
}
```

Our extractor reported a site in the `return` statement, while Frama-C reported the `printLine()`, where the invalid address is used. Both make sense. Since our definition of site was local, in order to automatically handle Frama-C warnings, the sites for this class has to be *use* of returned value. Only two test cases had examples of this condition, so we checked them manually.

Results:

1838 sites ($|U|$). 2 warnings ($|W|$). 1836 findings ($|F|$). 2 buggy sites ($|B|$).

We confirmed the following.

Warnings and sites were consistent: $W \subseteq U$

There were many findings: $|F| \geq 0.6 \times (|U| - |B|)$

All findings were correct: $F \cap B = \emptyset$

For CWE-562, Frama-C satisfied the criteria.

3.5. Warnings Handled As Exceptions

In many instances, Frama-C produced warnings that did not fit into the classes above or were not readily handled by our automated processing. We handled these warnings as exceptions.

Frama-C correctly warned about read after end of buffer flaws in 72 files. See Sec. A.5 in the appendix for more detail. Since these were unintentional, we decided not to take the time to change code to process these automatically.

We excluded 76 test cases, consisting of a total of 112 files, because they use `RAND32()`. Frama-C's model results in any execution after that use to be undefined. See Sec. 3.3.1 for more explanation. Since we could not easily determine what occurs in the control flow before `RAND32()` is used, we excluded these test cases altogether.

Frama-C warned about integer overflow for many uses of left shift (<<) in `RAND32()` and `RAND64()`. These are legitimate warnings, in that their behavior is undefined according to the C11 standard. Since they do not correspond to any of the above weakness classes, we excluded these 2101 warnings from automated processing.

Frama-C produced 152 "invalid memory access" warnings, specifically invalid write, for `calloc()` when the allocation fails. We doubt that actual library code tries to zero memory if allocation fails, so we considered these warnings to be model artifacts.

3.6. Errors in Juliet that Frama-C Found

This section explains the three previously-unknown systematic errors in Juliet 1.2 that Frama-C's warnings uncovered.

In addition to the three systematic errors, Frama-C warned about constructs that occurred in four test cases. In those four cases, the program stored a value in one member of a union, then read from the other member, a process sometimes called "type punning." This is allowed by the ISO/IEC C 2011 standard. There are more details in Sec. A.4 in the appendix.

3.6.1. Minimal Read After End of Buffer. 72 files in Juliet had an unintentional bug, see Sec. A.5 in the appendix. The source string passed to `memcpy()` or `memmove()` was shorter than the size to be copied or moved. In this cases, Frama-C correctly pointed out a minimal read after end of buffer for the function call. Since these flaws were unintentional, we excluded them as sites in Ockham Criteria.

3.6.2. Use of Object Past Its Lifetime. In the set of warnings for "making use of address of object past its lifetime," Frama-C warned of a previously unnoticed systematic problem in the Juliet set. It is use of memory after its lifetime, which produced 152 warnings in 140 test cases. Here is an example of the code from `CWE476_NULL_Pointer_Dereference__int_34.c`, SARD case 104 717:

```
int * data;
{
    int tmpData = 5;
    data = &tmpData;
}
printIntLine(*data);
```

The address of `tmpData` is passed to `printIntLine()` after its life time. There are more details in the Appendix, Sec. A.7.

Class (Related CWEs)	Sites ($\|U\|$)	Warnings ($\|W\|$)	Findings ($\|F\|$)	Buggy Sites ($\|B\|$)
Write Outside Buffer (121, 122, 124)	97 678	18 767	78 911	7400
Write-what-where (123)	72 084	791	71 293	228
Read Outside Buffer (126, 127)	66 665	3396	63 269	2168
NULL Pointer Dereference (476)	72 084	303	71 781	271
Integer Overflow (190)	29 907	1356	28 551	1026
Divide by Zero (369)	3 018	1399	1 619	684
Uninitialized Variable (457)	263 520	770	262 750	560
Return Stack Variable (562)	1 838	2	1 836	2

Table 1. Number of Sites, Warnings, Findings, and Buggy Sites For Each Weakness Class

3.6.3. Uninitialized Struct Member. In 204 test cases a mistake failed to initialize the second member of a structure. Here is a version of the pertinent code from `CWE121_Stack_Based_Buffer_Overflow__CWE805_struct_declare_loop_01.c`, SARD case 64 912. The `typedef` comes from `std_testcase.h`.

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

    twoIntsStruct source[100];

    for (i = 0; i < 100; i++)
    {
        source[i].intOne = 0;
        source[i].intOne = 0;
    }

    for (i = 0; i < 100; i++)
    {
        data[i] = source[i];
    }
```

In other examples of this pattern, `intTwo` is initialized also instead of `intOne` being initialized twice. Frama-C warned of "reading from uninitialized lvalue" noting that `intTwo` should be initialized.

All the test cases are in the subdirectory `CWE121_Stack_Based_Buffer_Overflow`. There are more details in Sec. A.6 in the appendix.

3.7. Summary of Evaluation

This section summarizes the evaluation of Frama-C on the SATE V Ockham Sound Analysis Criteria. First, we summarize all the considerations involved in the evaluation. Next, we summarize the number of sites and the final result of the evaluation.

3.7.1. Summary of Considerations. The August 2013 development version of Frama-C, which was used for the Ockham Criteria, did not support wide string literals, e.g. `L"Good"`, nor the format specifier for wide string (`%ls`).

Frama-C terminates analysis when it detects certain failures. Therefore there are no sites for Ockham purposes after a terminating failure. Section 3.1.1

Warnings are the union of two sets of runs: one that assumed that every allocation failed, and one that assumed that every allocation succeeded. Section 3.1.1

Frama-C always warns about buggy sites, but may warn about sites without bugs. Therefore a finding is a good site, that is, a site without a warning. Section 3.1.1

We determined which sites are buggy from the structure of Juliet code, by manual inspections, and by reconciliation with Frama-C warnings. Section 3.1.1

Warnings in test cases that use on the `RAND32()` macro were not included, as explained in Sec. 3.3.1.

Because Frama-C was not run on CWE191 test cases, we excluded all sites under the CWE191 subdirectory from the universe. Section 3.3.2

If `memcpy()` or `memmove()` write into a structure, it is not a site of a buffer write. Section 3.4.1

We grouped warnings for CWE-121, CWE-122, and CWE-124 into the same class, which we named Buffer Write. Section 3.4.1 Similarly, we grouped warnings for CWE-126 and CWE-127 into the Buffer Read class. Section 3.4.3

We only checked warnings in the CWE476 subdirectory for NULL pointer dereference. Section 3.4.4

Frama-C only reports integer overflow in data types larger than `int`. Section 3.4.5

Warnings for `invalid arguments to library function` were not mapped to any CWEs at the beginning of the analysis. We only included them when Frama-C did not produce any other warning for a buffer overread. These warnings were reported as occurring in input/output utility functions in `io.c`. We extracted the name of the file where the utility function was called from additional information in the warning. It means that for some of the files cited are not found in Frama-C xml output "location" tag.

3.7.2. Summary of Results. The number of sites, warnings, findings, and buggy sites for each class is given in Table 1. In the test cases selected from the Juliet 1.2 test suite, we considered a total of 606 794 sites in eight classes of weaknesses. There were a total of 12 339 buggy sites. Counting the excluded and the unclassified warnings, which are not listed above, we processed a total of 31 955 unique Frama-C warnings.

Frama-C satisfied the SATE V Ockham Sound Analysis Criteria.

4. Future Changes

This section suggests changes for future Ockham Criteria.

4.1. Weakness Classes

Although the SATE V Ockham Sound Analysis Criteria used the term "weakness classes," the classes are not specified. We had CWE classes in mind. In most cases Frama-C used classes of warnings that did not correspond well to CWEs. For instance, Frama-C did not distinguish between CWE-121 Stack-based Buffer Overflow, CWE-122 Heap-based Buffer Overflow, and CWE-124 Buffer Underwrite. As we proceeded in the analysis, we did not see much benefit in holding rigidly to CWE distinctions.

In general, weakness classes that tools use only approximately match CWE classes, see Sec. 2.4 in [11].

We spent a lot of time mapping tool warnings to our pre-conceived classes. Within classes of tool warnings, we matched some of the warnings in a tool class to one CWE-based class and other warnings in the same tool class to other classes derived from CWEs. It may have been easier to just evaluate the warnings as given. In either case, we had to try different matches and communicate with the tool developers to understand what class of weaknesses the tool was reporting.

Without understanding what class of weakness the tool was considering, we could not decide whether a buggy site corresponded to a tool warning. (This happened many times during analysis.) If the tool was designed to cover that class, then a mismatch indicated a missed buggy site and 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 buggy site should be ignored. In all our analysis, we concluded that our notion of a class, and hence sites for the class, needed to correspond with the class that the tool actually checked.

In the future, we plan to use the weakness classes that the tools use.

For ease of information sharing, we are researching a more universal approach to characterizing weakness classes.

4.2. Definition of Site

As mentioned in Sec. 2.2, it is not always clear what location in a flow of execution should be considered to be a site. For instance, a function may have a few lines of code to copy a string, which have sites of read buffer and write buffer. If the code instead calls the standard library function `strcpy()`, the situation changes. If the only sites are considered to be within the body of `strcpy()`, then thousands of invocations throughout the code base appear to condense into a few places. In addition, the source code is probably not available.

A better definition may be that a site is the final place that the programmer can make any checks that are necessary or arrange the state properly. When the programmer invokes a standard function or uses a built-in operator, the programmer must satisfy their preconditions. This may justify declaring that sites are in the main line code.

The question is still open as to what should be declared to be the site of missing code, such as failure to check user input.

4.3. Number of Findings

Criteria 2 stated that the tool produces findings for a minimum of 60 % of applicable sites. The limit, 60 %, was somewhat arbitrary. The minimum percentage of findings per good (non-buggy) site was 69 % for CWE-369 Divide by Zero cases. We attribute this relatively low percentage of findings to Frama-C's implementation choices for abstract interpretation, as explained in Sec. 2.3.2.

The next lowest percentage of findings was 87 % for Write Outside Buffer. For all other weakness classes, Frama-C produced findings for nearly 100 % of the good sites. We note that the Juliet test cases are synthetic cases and do not have the complexity found in typical production code.

A minimum of 75 % may be a reasonable limit for the next Ockham.

4.4. No Errors

Criteria 3 stated that even one incorrect finding disqualifies a tool. For a production piece of code, this may be overly demanding. A tool built with a known architectural shortcoming or an inference engine with theoretical limitations could *never* achieve a flawless evaluation, regardless of the care or amount of debugging that is done. However, there are many, many details to specify the semantics of a programming language and libraries and to encode policies and definitions. Tools can still be very useful if there is one minor mistake that is easily fixed, as opposed to a systematic difficulty that requires major reengineering to overcome.

Perhaps the next Ockham should require no *pattern* of incorrect findings or incorrect findings resulting from structural or theoretical reasons.

4.5. Use of the Term “Sound”

As explained in Sec. 2.3, the SATE V Ockham Criteria used the term “sound” and “complete” in almost the reverse sense of that used by a large, well-established formal methods community and their considerable body of published work. Although Ockham’s use may have been reasonable, it would cause unnecessary and unproductive confusion for the terms to be used very differently in similar contexts. Trying to change the community’s use would require a huge effort for a relatively small gain.

Future Ockham Criteria should adopt a term other than “sound.” Some possibilities are “correct,” “flawless,” “reliable,” “faithful,” “faultless,” or “exact.”

5. Conclusions

Pascal Cuoq and Florent Kirchner ran the August 2013 development version of Frama-C on 13 706 test cases from the Juliet 1.2 test suite. This produced a total of 31 955 unique warnings covering over half a million sites.

The reports from Frama-C led to the discovery of three kinds of unintentional, systematic flaws in the Juliet test suite. These flaws involve 416 test cases. For flaws with straightforward fixes, we will replace the flawed test cases by fixed versions in the SARD [5].

The version of Frama-C that was used, the August 2013 development version, did not support wide string literals, e.g. `L"Good"`, nor the format specifier for wide string (`%ls`).

Frama-C satisfied the SATE V Ockham Sound Analysis Criteria.

Acknowledgments

We thank Yaacov Yesha and Irena Bojanova for their extensive comments, which greatly improved the paper. We also thank Charles de Oliveira and Christopher Long for their work in analysis. We are particularly indebted to Pascal Cuoq and Florent Kirchner, who ran Frama-C and answered many questions about interpreting the result.

References

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

[2] (2014) Static Analysis Tool Exposition (SATE) V. [Online]. Available: http://samate.nist.gov/SATE5.html

[3] (2015) Rice’s theorem. [Online]. Available: http://en.wikipedia.org/wiki/Rice’s_theorem

[4] T. Boland and P. E. Black, “Juliet 1.1 C/C++ and Java test suite,” *IEEE Computer*, vol. 45, no. 10, pp. 88–90, Oct 2012.

[5] (2016) Static Assurance Reference Dataset (SARD). [Online]. Available: http://samate.nist.gov/SARD/

[6] N. Mansourov and D. Campara, *System Assurance: Beyond Detecting Vulnerabilities*. Morgan Kaufmann, 2010, pp. 177–178.

[7] P. A. Laplante, Ed., *Dictionary of Computer Science, Engineering, and Technology*. CRC Press, 2001, pp. 90, 459.

[8] (2014) What is Frama-C. [Online]. Available: http://frama-c.com/what_is.html

[9] (2016) XZ Utils. [Online]. Available: http://tukaani.org/xz/

[10] “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, Tech. Rep., 2011.

[11] V. Okun, A. Delaitre, and P. E. Black, “Report on the static analysis tool exposition (sate) IV,” National Institute of Standards and Technology, Special Publication 500-297, January 2013, http://dx.doi.org/10.6028/NIST.SP.500-297.

Appendix A. Details of SATE V Evaluation

This appendix includes details about specifics of the evaluation of SATE V Ockham Sound Analysis Criteria. Details include the names of specific test cases and names from Juliet 1.2, C code, extended explanations, and warnings.

The following description of the Juliet 1.2 test suite draws heavily from and quotes Boland and Black [4]. The Juliet 1.2 test suite is a collection of C/C++ and Java programs with known flaws. “Each program or test case consists of one or two pages of code . . . The test cases are synthetic, that is, they were created as examples with well-characterized weaknesses.” Each case is intended to exhibit only one flaw.

Test cases are organized by the most similar CWE weakness class. Thus the subdirectory `CWE121_Stack_Based_Buffer_Overflow` only has cases that test write after end of buffer in the stack. The subdirectories that have thousands of test cases have up to nine subdirectories named `s01`, `s02`, `s03`, etc.

Each C test case comprises one or more files with names such as `CWE134_Uncontrolled_Format_String__char_file_printf_22a.c`. The file name has the following components: a CWE number and short name, a functional variant (`char_file_printf`, in this case), a two-digit structure number (22), an optional subfile indicator (a), and the extension .c.

Functional variants name data types, library functions, or structures. Flow structure numbers indicate the type of data or control flow used, for example, loop, data flow, local control flow, constant in conditional, passing data by a function call, data type, container, etc.

In addition to files with shared declarations and common utilities, a test case can consist of one source code file or of multiple files. For example, test case `CWE476_NULL_Pointer_Dereference__char_01` is contained in one source code file (in addition to shared files). In contrast `CWE23_Relative_Path_Traversal__wchar_t_connect_socket_w32CreateFile_54` has five files–`54a.c`, `54b.c`, through `54e.c`–that constitute one test case.

A.1. Frama-C Not Run on CWE191 Cases

Because of a simple human mistake, Frama-C was not run on the CWE191 test cases. The beginning of the main script used to generate run Frama-C on the test cases is shown in Fig. 1. The script processes all the test cases that were in the directory `testcases/CWE191_Integer_Underflow`. Unfortunately, there were no test cases right in that directory; all the test cases were in the `s0*` subdirectories `testcases/CWE191_Integer_Underflow/s0*`.

For consistency, we excluded all sites under the CWE191 subdirectory from the universe.

The developers later submitted files with the warnings. We decided not to evaluate those files since they came from a later version of Frama-C.

A.2. Frama-C Does Not Handle Some Wide Characters in `printWLine()`

Frama-C did not handle wide characters passed to the utility function `printWLine()` in 36 test cases. The test cases are named `CWE126_Buffer_Overread__CWE170_wchar_t_strncpy_01.c` through `18.c` and `_memcpy_ 01.c` through `18.c`.

Briefly, the wide string format specifier, `%ls`, was not modeled properly. Here is an example of code from Juliet.

```
wchar_t data[150], dest[100];
wmemset(data, L'A', 149);
data[149] = L'\0';
memcpy(dest, data, 99*sizeof(wchar_t));
printWLine(dest);
```

As written, the string that is placed in `dest` is not null terminated because only 99 wide characters are copied.

The problem was that `wmemset()` fills the array with alternate bytes containing character `A` and a null. In Frama-C the wide string format specifier was modeled as taking one byte at a time. Frama-C’s analysis found that taking a byte at a time, a null character was encountered. Therefore no warning was reported.

We excluded sites in these cases. This exclusion code is in `extractor.py`. Search for the string `printWLine() sites` to find it.

A.3. Mismatched Buggy Sites Excluded Because of Undefined Behavior

As explained in Sec. 3.3.1, the Juliet 1.2 test suite uses a macro that Frama-C determines to yield undefined behavior. The macro is as follows:

```
#define RAND32() \
  ((rand()<<30) ^ (rand()<<15) ^ rand())
```

The International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC) C 2011 standard Sec. 6.5.7 Bitwise shift operators, paragraph 3 says, “If the value of the right operand is negative or is greater than or equal to the width in bits of the promoted left operand, the behavior is undefined.” [10]

```
for DIR in \
    testcases/CWE190_Integer_Overflow/s* \
    testcases/CWE191_Integer_Underflow \
    testcases/CWE457_Use_of_Uninitialized_Variable/s* \
    testcases/CWE123_Write_What_Where_Condition \
    ...
```

Figure 1. Beginning of Main Script to Run Frama-C Showing Error in Accessing CWE191 Cases

Frama-C models `rand()` as returning a type that is less than 30 bits. According to the standard, the result of the expression is undefined. Note that in C, "undefined" is more drastic than "the result may be any number." The term "undefined" means that following execution of a statement with `RAND32()`, the program can whistle "Happy Birthday" in all the colors of the rainbow and still be considered to conform to the standard.

Frama-C handles undefined behavior in an execution by not doing any further analysis. Hence the refinement about terminating failures in Sec. 3.1.1.

Our site extraction is largely syntactic or local. We do not track the more than a few statements of the flow of execution. Hence our automated extraction would otherwise include some buggy sites that follow undefined behavior. The most difficult cases use `RAND32()` in one file then call a function in another file, where the weakness is. Here is the essence of the code, from Software Assurance Reference Dataset (SARD) [5] case 74 230. The file `CWE124_Buffer_Underwrite__CWE839_rand_68a.c` has

```
void CWE124_...rand_68_bad()
{
    data = RAND32();
    CWE124_...rand_68b_badSink();
}
```

while the file `CWE124_Buffer_Underwrite__CWE839_rand_68b.c` has

```
void CWE124_...rand_68b_badSink()
{
  buffer[data] = 1;// POSSIBLE UNDERWRITE
}
```

The variable `data` is a global variable.

The thorough approach would be to carefully follow extraction flow and exclude all sites following the use of `RAND32()`.

We adopted the most principled approach given our basic checking analysis. This approach was to completely exclude the 76 test cases that use `RAND32()`.

For Write Outside Buffer, we excluded the 38 test cases with the prefix `CWE124_Buffer_Underwrite__CWE839_rand_`. These 38 test cases comprise a total of 56 files. Some test cases exercise inter-file calls, so have name suffixes like `52a.c`, `52b.c`, and `52c.c`. The 38 test cases are suffixes `01` through `18`, `21`, `22`, `31`, `32`, `34`, `41`, `42`, `44`, `45`, `51` through `54`, `61`, and `63` through `68`.

For Read Outside Buffer, we excluded the 38 test cases with the prefix `CWE127_Buffer_Underread__CWE839_rand_`. The pattern of test cases and files exactly follows that of Write Outside Buffer.

Frama-C also warned about integer overflow for the macro `RAND64()`. There are 114 cases that use `RAND64()`. They start with `CWE190_Integer_Overflow__int64_t_rand_`, which is followed by `add`, `multiply`, or `square`. The specific cases are `01` through `18`, `21`, `22`, `31`, `32`, `34`, `41`, `42`, `44`, `45`, `51` through `54`, `61`, and `63` through `68`. We did not find that these warnings caused any problem with analysis.

A.4. Incompatible Access Types Warnings

Frama-C reported that four of the cases with previously unknown errors of use of memory after lifetime, as noted in Sec. A.7, had an additional error. These cases, ending in "34," store to one member of a union, then read from a different member, a process sometimes called "type punning." Here is an example of the code, from `CWE476_NULL_Pointer_Dereference__int_34.c`, SARD case 104 717:

```
typedef union
{
    int * unionFirst;
    int * unionSecond;
} CWE476_...int_34_unionType;

    CWE476_...int_34_unionType myUnion;
    {
        int tmpData = 5;
        data = &tmpData;
    }
    myUnion.unionFirst = data;
    {
        int *data = myUnion.unionSecond;
        printIntLine(*data);
    }
```

The ISO/IEC C 2011 standard 6.5.2.3 Structure and union members, footnote 95 says, “If the member used to read the contents of a union object is not the same as the member last used to store a value in the object, the appropriate part of the object representation of the value is reinterpreted as an object representation in the new type . . . (a process sometimes called “type punning”).” [10]

This construct is well defined in the C 2011 standard. However, since other versions of the standard are not clear about how it should be treated, we believe that Frama-C was reasonable to model this as incompatible access type.

A.5. Previously Unknown Error: Unintended Minimal Read After End of Buffer Bug

72 Juliet test cases have an unintentional bug that is minimal read after end of buffer. The pertinent code is

```
#define SRC_STR "0123456789abcde0123"

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

charVoid structCharVoid;
```

followed by one of the following two lines

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

memmove(structCharVoid.charFirst,
      SRC_STR, sizeof(structCharVoid));
```

The problem is as follows. `SRC_STR` is 20 characters (bytes) long, including the null terminator. `sizeof(structCharVoid)` is at least 24 bytes long: 16 characters (bytes) in `charFirst` and 4 bytes for each of the two pointers. The standard functions `memcpy()` and `memmove()` therefore read at least 24 bytes from the constant string that is only 20 bytes.

Both `memcpy()` and `memmove()` allow *writing* beyond the end of a buffer. (They are often used to copy or initialize an entire structure when only the first field of the structure is passed.) Thus the intended bug, write outside buffer, is not present given standard semantics.

The 72 files with these problems are SARD [5] test cases 63 036 to 63 071 and 67 448 to 67 483. All of the file names begin with `CWE121_Stack_Based_Buffer_Overflow__`. Next in the name is the data type. (In the Juliet collection, the `char_ type_` files are in subdirectory `s01`, and the `wchar_t_ type_` files are in subdirectory `s09`.) The name next has the operation, which is `overrun_memcpy_` or `overrun_memmove_`. There are 18 control flow variants, `01.c` through `18.c`, for each. As an example, here is the complete name of one file, which is in subdirectory `s09`: `CWE121_Stack_Based_Buffer_Overflow__wchar_t_type_overrun_memmove_13.c`.

A.6. Previously Unknown Error: Uninitialized Storage

In 204 test cases a mistake failed to initialize the second member of a structure. Here is a basic version of the pertinent code from `CWE121_Stack_Based_Buffer_Overflow__CWE805_struct_declare_loop_01.c`, SARD [5] case 64 912. The `typedef` comes from `std_testcase.h`.

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

    twoIntsStruct source[100];

    for (i = 0; i < 100; i++)
    {
        source[i].intOne = 0;
        source[i].intOne = 0;
    }

    for (i = 0; i < 100; i++)
    {
        data[i] = source[i];
    }
```

In other examples of this pattern, `intTwo` is initialized also instead of `intOne` being initialized twice. Frama-C warns of “reading from uninitialized lvalue” noting that `intTwo` should be initialized. (In the `memcpy()` and `memmove()` cases, `source` is read by the respective function, not a primitive assignment.)

All the test cases are in `CWE121_Stack_Based_Buffer_Overflow` and begin with that string. The first 69 cases are in subdirectory `s04`; the rest are in subdirectory `s05`.

The test cases came in 2 functional variants (`CWE805_struct_alloca` and `declare`) × 3 subvariants (`loop`, `memcpy`, and `memmove`) × 34 templates (`01` through `18`, `31`, `32`, `34`, `41`, `44`, `45`, `51b`, `52c`, `53d`, `54e`, and `63b` through `68b`) = 204 test cases. For example, the first warning is in `CWE121_Stack_Based_Buffer_Overflow__CWE805_struct_alloca_loop_01.c`, SARD case 64 792, and the last warning is in `CWE121_Stack_Based_`

Buffer_Overflow__CWE805_struct_declare_memmove_68b.c, SARD case 65 026. (Some of the cases in that span do not have this bug.)

A.7. Previously Unknown Error: Use of Memory After Lifetime

Frama-C warned of a previously unnoticed systematic mistake in the Juliet set. In the supposedly good versions of NULL pointer dereference cases, the following type of code occurs in many variations. This comes from CWE476_NULL_Pointer_Dereference__int_01.c, SARD case 104 694:

```
int * data;
{
    int tmpData = 5;
    data = &tmpData;
}
printIntLine(*data);
```

The variable tmpData is declared in an inner scope. The lifetime of its memory ends at the closing bracket. Thus the dereference in printIntLine() is undefined behavior. However, most compilers do not release the memory until the end of the enclosing function, so the code usually works fine. The ISO/IEC C 2011 standard Sec. 6.2.4 Storage duration of objects, paragraph 6 says, "For such an object ... its lifetime extends from entry into the block with which it is associated until execution of that block ends" [10] Frama-C's warning is consistent with this.

These mistakes in Juliet could be labeled as CWE-825 Expired Pointer Dereference.

There are 152 warnings in 4 versions (int64_t_, int_, long_, and struct_) x 35 templates (01 through 18, 21, 22a, 31, 32, 41, 44, 45, 51a through 54a, 63b, 64b, 65a, 66b, 67a, and 68b) = 140 test cases. Each template ending in "12" had four warnings.