# Software Model-Checking: Benchmarking and Techniques for Buffer Overflow Analysis

by

Kelvin Ku

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

# Abstract

Software Model-Checking: Benchmarking and Techniques for Buffer Overflow Analysis

Kelvin Ku

Master of Science

Graduate Department of Computer Science

University of Toronto

2008

Software model-checking based on abstraction-refinement has recently achieved widespread success in verifying critical properties of real-world device drivers. We believe this success can be replicated for the problem of buffer overflow detection. This thesis presents two projects which contribute to this objective. First, it discusses the design and construction of a buffer overflow benchmark for software model-checkers. The benchmark consists of 298 code fragments of varying complexity capturing 22 buffer overflow vulnerabilities in 12 open source applications. We give a preliminary evaluation of the benchmark using the SatAbs model checker. Second, the thesis describes the implementation of several components for supporting buffer overflow analysis in the YASM software model-checker.

# Contents

# Chapter 1

# Introduction

## 1.1 Overview

The buffer overflow vulnerability is a serious security concern and has been studied in detail as early as 1995 [43]. A buffer overflow occurs when a program writes data into a fixed-size array past the end of the array and overwrites adjacent data. This typically occurs in C programs which use unchecked string library functions to copy data into arrays. If the data being written is user input, an attacker can exploit the vulnerability by including machine instructions in the input which the program subsequently executes. Thus, the vulnerability enables an attacker to use a host program to execute arbitrary code. If the program is running with root privileges, this effectively gives the attacker control of the host system.

Since the vulnerability is such a serious security threat, there is a strong motivation for detecting and eliminating potential buffer overflows in a given program before it is deployed. Static analysis is a promising approach to this end and there has been much research in techniques for efficiently detecting potential vulnerabilities in C programs [49, 27, 51, 53]. However, buffer overflow analysis is an inherently hard problem. To see why, consider the simple C program shown in Figure 1.1(a). The example shows a typical

```
void main(void)
{
  char input[10], buf[9], *src, *dest;

  dest = buf;
  src = input;
  while (*src)
  {  VULN: *dest++ = *src++; }
}
```

(a)

```
void main(void)
{
  char input[10], buf[9], *src, *dest;

  input[8] = '\0';

  dest = buf;
  src = input;
  while (*src)
  { VULN: *dest++ = *src++; }
}
```

(b)

Figure 1.1: Example programs illustrating the difficulty of buffer overflow analysis: vulnerable (a), safe (b).

string copy loop which iterates over the string `input`, copying each character into the array `buf`. To show that the statement labelled "`VULN`" could potentially overflow the array `buf`, an analysis needs to show that there is a value of `input` for which the loop executes at least ten times. Since `input` is uninitialized, it could have any of $256^{10}$ possible values. Now consider the example in Figure 1.1(b). The addition of the single statement `input[8]` $=' \backslash 0'$ has removed the vulnerability from the original example, thereby making it *safe*. An effective analysis should distinguish between the two cases, i.e., it should show that the second case is safe. However, this is no easier than the original problem, since it requires showing that the loop executes at most 9 times for *all* values of `input`. Note that these examples are drastic simplifications of code that appears in real-world programs which are typically much larger (e.g., 10K-1M lines of code) and contain complex syntax and structure.

The goal of our work is to develop an effective static analysis for buffer overflows. We believe such an analysis should provide a high overflow detection rate with few false alarms and should efficiently handle real-world programs. Specifically, we define four basic analysis requirements:

1. **Soundness.** The analysis should only produce correct results. It should not indicate a vulnerability where one does not exist.

2. **Verification.** The analysis should be able to prove that a program is free of

vulnerabilities. Users should be able to check the correctness of a patch for a previously detected vulnerability, such as the addition of the statement `input[8] = '\0'` in Figure 1.1(b).

3. **Verifiable results.** The analysis should produce output which can be compared to the input program in order to verify the results. Without this output, it is possible for a tool to (1) produce incorrect results without the user knowing or (2) produce correct results for a given program because of incorrect reasoning.

4. **Robustness and scalability.** The analysis should accept real-world programs which may contain complex syntax and many lines of code.

## 1.2 Techniques for Static Analysis of Buffer Overflows

In this section we provide an overview of current techniques for static analysis of buffer overflows.

*Testing.* The basic formulation of testing is to repeatedly run a program on randomly generated inputs in an attempt to trigger a buffer overflow. Although testing has been successfully applied to finding several buffer overflows [38], it performs poorly when it must navigate through many branches [29]. *Directed testing* [45] improves upon random testing by using symbolic execution to generate inputs which drive execution down previously unexplored paths, thus improving the coverage of the tests. Testing-based techniques satisfy all but one of our requirements, namely, verification, since it is infeasible to test all possible executions of a program. Likewise, testing may require an unacceptable amount of time to discover a vulnerability if one in fact exists, due to the intractable number of inputs that must be tested.

*Syntax-directed techniques.* This is a family of techniques which identify patterns in

source code which may indicate a vulnerability. Tools such as ITS4 [48] and RATS [51] search for common potentially dangerous uses of standard library functions, such as `printf("%s", in)` where the variable `in` holds user input. These techniques suffer from *false alarms*: they may flag a particular use of a function as unsafe even though some other part of the program ensures that the use is in fact safe. Likewise, they may fail to detect a potential vulnerability if it does not match one of the pre-specified patterns. In general, these tools can only provide a very limited degree of assurance to the user.

*Semantics-directed techniques.* This is a class of techniques based on a variety of static analyses such as dataflow analysis, abstract interpretation, and constraint analysis. These techniques typically construct a simple abstraction of a program on which some limited class of properties can be checked efficiently. The abstraction only models a subset of a program's semantics, so the subsequent analysis is subject to both missed detections and false alarms. Tools such as Boon [49] (based on interval abstraction) and Splint [27] (based on linear constraint analysis) have been shown to perform poorly at buffer overflow detection and verification [53].

The discussion below concentrates on the static analysis technique known as software model-checking. It is a promising approach to buffer overflow analysis and has the potential to overcome the limitations of the techniques discussed above.

### 1.2.1 Software Model-Checking

Model-checking is a technique for verifying temporal properties of a finite-state machine (i.e., a *model*) [20]. Software model-checking (SMC) builds on this to analyze *programs* by automatically constructing a model which preserves certain behaviours of a given program and using a model-checker to verify a specified property of the program, e.g., that an assertion is never violated. There are a variety of approaches to software model-checking which we summarize below.

*Explicit-state* SMCs such as Java PathFinder [32] construct a program model on-the-

fly by executing the program in a virtual machine. At each execution step, it stores a small record of the current machine state, such as a hash of the program counter and memory, which it uses to avoid re-executing previously encountered states. At each control-flow branch, it attempts to choose values for non-deterministic data, such as user input and thread scheduler variables, that will drive execution to a desired state (e.g., an error state). The process continues until resources are exhausted or a desired state is found.

*Bounded* SMCs construct a model representing all behaviours of a program for a finite number of steps. Loops and recursive functions calls are unwound to a specified bound and the result is encoded as a SAT formula or Binary Decision Diagram (BDD) which is conjoined with the specified property. A decision procedure such as a SAT solver is then used to check the property. CBMC [22] is a bounded model-checker for C that uses a SAT solver to check user-specified assertions in a given program.

*Counterexample-guided abstraction-refinement (CEGAR)* SMCs such as Blast [16] and Slam [15] iteratively construct an abstract model of a given program using *predicates* (boolean expressions over program variables) to represent program state. The process begins with the program's control-flow graph as the model. Each state of the model initially represents no knowledge about the program state at each control-flow location. The model is then checked for a specified property; suppose we are checking the reachability of an error state. If the model-checker fails to find a path to the error state, the process terminates and the program is deemed safe with respect to the error. Otherwise, it returns a *counterexample*: a path through the model terminating in the error state. The counterexample is validated against the original program to see if each step in the path is feasible. If it represents a real execution of the program, then it is returned to the user as proof of an error. If not, it is an artifact of the overly coarse model of the program. In this case, predicates tracking the relationships between certain program variables (e.g., those which control the reachability of the error state) are added to the

model and the process is repeated.

SLAM [12] is the canonical CEGAR SMC; it uses a theorem-prover to construct the abstract model and a symbolic executor to perform counterexample validation. Since the development of SLAM, there have been a number of refinements to the CEGAR design. Blast [16] attempts to improve upon the standard approach by constructing a model containing only the reachable states of the system. SatAbs [22] uses a SAT solver in place of a theorem-prover, thereby obtaining a much faster abstraction process. Copper [18] uses a specialized abstraction for standard C library functions which enables it to efficiently analyze certain uses of these functions. Finally, YASM [31] uses multi-valued logic to construct a more precise model in which non-determinism is explicitly represented and counterexamples do not need to be validated.

### 1.2.2 Software Model-Checking: Current Status

Explicit-state SMCs have been shown to be effective in buffer overflow *detection* [17]. However, since this technique is limited to exploring finitely many behaviours of a given program, it is unsuitable for *verification*, i.e., for proving properties about all behaviours. Bounded SMCs may be used for verification but only if a sufficient bound can be found [21]. This technique has been found to have a significant performance dependence on the bound size [9], so its scalability in real-world programs is unclear.

In theory, CEGAR SMC satisfies the first three requirements of an effective analysis. It is sound since each potential counterexample is validated before being reported to the user. It may be used for verification since the analysis only terminates when it has eliminated all counterexamples from the model; it may also in some cases construct a safe model in a few iterations. Finally, when CEGAR SMC finds a vulnerability, it produces verifiable results in the form of a counterexample.

In the published evaluations of CEGAR SMC implementations, such as those of Slam [12], Blast [33], and SatAbs [23], the tools have been shown to be effective in

checking user-specified assertions in programs such as Windows and Linux device drivers and open-source applications with up to tens of thousands of lines of code. Due to its success in these domains, we believe CEGAR SMC is a promising technique for analysis of large-scale, real-world code.

### 1.2.3 CEGAR SMC: Limitations

There are several outstanding issues concerning CEGAR SMC and buffer overflow analysis. First, it is well known that CEGAR has a significant performance dependence on the size of the loop bounds in the given program [29, 36, 37]. Buffer overflows often involve loops over very large arrays (e.g., 512 or 4096 elements), so this dependence could severely limit the utility of CEGAR in this domain. Second, we are unaware of any significant studies of the real-world performance of CEGAR SMCs in buffer overflow analysis. The only evaluation we found is in [37], in which the performance of three CEGAR SMCs is compared on a single ten line buffer overflow example. Thus, we lack empirical knowledge regarding the suitability of CEGAR in this domain. Finally, it is unclear why CEGAR performs well in analyzing certain types of programs, such as device drivers, and whether this success can be replicated for buffer overflow analysis. Specifically, we lack an understanding of the relationship between the structure of (programs containing) buffer overflows and CEGAR performance.

## 1.3 Objectives

The goal of this thesis is to support the development of CEGAR SMC as an effective static analysis for buffer overflows. We approach this goal in two parts. First, by developing a buffer overflow benchmark for CEGAR SMCs. A benchmark is a natural method both for evaluating the fitness of CEGAR for buffer overflow analysis and for understanding the relationship between analysis complexity and problem structure. Second, by extending

YASM with the basic functionality required for buffer overflow analysis. The purpose of this is to establish YASM as a basis for further development in this area. Our specific objectives are as follows.

1. Determining the current state of CEGAR SMC with respect to buffer overflow analysis.
2. Constructing a buffer overflow benchmark suitable for evaluating CEGAR SMCs.
3. Understanding the relationship between program structure and analysis complexity.
4. Enabling basic buffer overflow analysis in YASM.

## 1.4 Organization

The thesis is organized as follows. Chapter 2 covers the design and evaluation of a buffer overflow benchmark for CEGAR SMCs. Chapter 3 describes YASM and the extensions for enabling buffer overflow analysis of real-world C programs. Chapter 4 concludes the thesis. Appendices include the raw data of the benchmark evaluation and source code for the testcase complexity measurement tool.

# Chapter 2

# Benchmarking Software Model-Checkers

In this chapter we discuss the design and implementation of a buffer overflow benchmark for software model-checkers. The chapter begins with basic definitions in Section 2.1. Section 2.2 defines the general objectives and specific requirements of the benchmark. Section 2.3 provides an overview of our benchmark development process. Section 2.4 discusses basic static measures for estimating testcase complexity. Section 2.5 describes our testcase construction process. Section 2.6 summarizes the contents of the benchmark. Section 2.7 documents an evaluation of the benchmark. Section 2.8 discusses related work. Finally, Section 2.9 concludes the chapter with a discussion of future work and lessons learned.

## 2.1 Benchmarking Background

Here we define the terms used in this chapter.

**Benchmark.** A set of tests for comparing the performance of a collection of tools on a common task. In this case, the tools of concern are CEGAR SMCs and the task is the verification of buffer overflows in C programs.

**Buffer overflow.** A buffer overflow occurs when a process accesses an address outside the bounds of allocated memory. A common case of this is where a program writes a string beyond the last element of an array and overwrites the adjacent cells. This is a security risk when the string is provided by the user and the adjacent data contains a program counter value which controls the subsequent execution of the process. Typically, the program counter appears as a return address or function pointer. To exploit an overflow, a user provides an input which overwrites the program counter to point to code which is included in the input. This enables the user to execute the supplied code with the privileges of the process.

**Vulnerability.** A program is vulnerable if there is an input for which the program overflows a buffer. The term *vulnerability* may refer to a vulnerable program or to a specific program statement in which a buffer overflow may occur. The simplest example of the latter is an array expression, e.g., `A[i]`. We also use the term *unsafe* to describe a vulnerable program and *safe* or *patched* to refer to a program which has had all known vulnerabilities removed.

**Common Vulnerabilities and Exposures (CVE).** A database of known security vulnerabilities in commercial and open-source programs. An entry in the database consists of a CVE identifier, e.g. CVE-2007-5381, and a description which typically includes the name and revision of the associated program and characteristics of the vulnerability.

**Testcase.** One or more source files comprising a single program. The source files contain distinguished statements to be checked for buffer overflows.

## 2.2 Benchmark Requirements

A benchmark for evaluating SMC performance in buffer overflow analysis serves two purposes. First, it enables SMC researchers to test analysis techniques in a realistic setting. Second, it provides a common basis for communicating research results. In

particular, for a single tool, it is used to measure (1) its performance profile across a range of testcases and (2) changes in performance between revisions of the tool. In the case of multiple tools, a benchmark is used to compare the relative strengths and weaknesses of the tools. Motivated by these objectives, we define several requirements for the benchmark as follows.

**R1: Realism.** Buffer overflows occur in a wide variety of programs, as indicated by the large collection of buffer overflow vulnerabilities in CVE (more than 4000). Likewise, the syntax and semantics, i.e., the *form*, of buffer overflows varies considerably across programs as well. On the other hand, results from the benchmark should strongly indicate the real-world performance of a given tool. Thus, the benchmark suite should (1) be comprised of testcases sampled from many types of program and (2) exhibit a variety of forms of vulnerabilities.

**R2: Verification and falsification.** As a corollary of R1, the suite should contain both safe and unsafe testcases. There are several reasons for this. First, the class of SMCs we are concerned with are designed for both verification and falsification (bug-finding). In a realistic setting, it cannot be assumed that a given program is safe or unsafe. As such, we assume that an SMC will be used for both purposes and, consequently, the testcases should elicit both types of analyses.

Second, since we are concerned with evaluating the soundness of SMCs, we need a method for determining if (1) a given tool obtains the correct result for a testcase and (2) whether it arrived at the result by correct reasoning. For a testcase known to be unsafe, determining (1) is straightforward: we simply check whether the tool finds the overflow. However, this does not ensure that the tool is sound, since, in the extreme case, it may simply return *unsafe* for all inputs. As such, each vulnerable testcase should be accompanied by a *patched* version in which the (known) vulnerabilities have been removed. If a tool produces the correct result for both the vulnerable and patched version of a case, it increases our confidence that the tool is not producing a correct

result as a by-product of a bug or an unsound heuristic.

**R3: Comprehensibility.** Each testcase should be short and simple enough for an unacquainted reader to understand its vulnerability and the accompanying patch. Constructing the benchmark this way makes it more accessible to others, since it frees users of the benchmark from having to work with real-world source code which is (1) not their primary concern and (2) unlikely to work correctly with prototype tools. This requirement is also related to our concern with tool soundness in R2. To check that a tool is reasoning correctly, we may want to compare its output, such as a counterexample, to the source code of a testcase—this requires reading and comprehending the source code.

**R4: Solvability.** Some fraction of the suite should be *solvable* by existing SMCs. That is, existing tools should be able to produce correct results for some number of testcases within reasonable time and memory constraints. If this fraction is too small, say, below 1/3, the benchmark produces too few results to evaluate the performance of a tool and the results mainly consist of timeouts or abnormal behaviour such. At this level of solvability, it is likely that the testcases are too complex for a research-grade tool to accept. If the fraction is too large, say, above 2/3, then the benchmark fails to challenge a tool and to indicate opportunities for improvement. At this level of solvability, it is likely that the testcases are too simple and the results mainly indicate that the tool finished in a negligible amount of time.

**R5: Configurability.** The benchmark should provide parameters with which the user can adjust the complexity of the testcases. This serves several purposes. The first is related to R4: setting parameters to lower levels may enable a tool to solve a larger portion of the testcases. Conversely, increasing parameter values makes the testcases more difficult, thereby exercising the scalability of a tool. Finally, analysis bottlenecks can be identified by varying each parameter independently and measuring the resulting change in performance.

## 2.3 Benchmark Development Process

This section describes the stages of our benchmark development process: initial assessment, defining complexity measures, testcase construction, benchmark evaluation, and evolution and maintenance. It provides an overview of the subsequent sections and motivation for each of the stages.

**Initial assessment.** This stage involves an informal evaluation of currently available SMCs. The purpose of the evaluation is to understand the current state of SMC with respect to buffer overflow analysis and to identify potential problems which a benchmark should accomodate.

For our preliminary experiments we chose to use the buffer overflow suite constructed by Zitser et al [53]. The tools we selected are all publically available SMCs for C programs: SLAM [14], Blast [16], Verisoft [28], SatAbs [24], CBMC [22], Copper [18], and YASM [31]. We found that all of the tools, with the exception of Copper, either produce incorrect results, crash, or fail to terminate on each of the testcases in the Zitser suite.

SLAM has been evaluated on real-world Windows device drivers written in C++, e.g., in [12]. However, it lacks support for arrays: it treats all array subscript expressions, e.g., `a[5]` and `a[i]`, as a single scalar variable. Also, its interface expects code to conform to the Windows device driver API and is thus unsuitable for verifying other classes of programs.

Blast has been used to check memory safety in C programs [16]. It too treats arrays in the same way as SLAM and thus cannot be used to analyze programs for buffer safety. Moreover, it requires programs to be preprocessed with CCured [42] since it lacks an automatic instrumentation facility. We found that CCured often generates unreadable code, making it difficult to compare the results of model-checking to the original code.

SatAbs has been used to check user-specified assertions in real-world C programs [22]. It includes a front-end which handles preprocessing, parsing, and instrumentation of the input code. Satabs crashed, either during parsing or analysis, on all of the testcases in

the Zitser suite. This is remarkable since the testcases are already simplifications of real programs, comparable in size and complexity to the example code which is distributed with this tool.

Copper has been evaluated on the Zitser suite [18]. It handles arrays in the same way as SLAM and Blast but uses a specialized abstraction to model standard C library functions, such as `malloc` and `strcpy`, which enables it to correctly analyze certain uses of these functions. On the Zitser suite, it detected 43% of the overflows and produced a false alarm in 21% of the cases. We found that Copper produces incorrect results for testcases in which the vulnerability depends on pointer arithmetic (array subscripting is an instance of pointer arithmetic).

YASM has been evaluated on several device driver examples distributed with Blast [31]. Chapter 2 describes extensions to YASM which enable basic support for pointers and arrays. It currently fails to parse the testcases in the Zitser suite and is limited to checking programs with small buffers.

Even after removing problematic syntax from the testcases to avoid crashing the tools, the tools failed to terminate on many of the cases before exhausting available resources. We also noticed that tool behaviour tended to be polarized: for a given testcase, a tool would either (1) crash or timeout or (2) solve the testcase in a negligible amount of time.

**Defining complexity measures.** In this stage we identify program features which have an impact on analysis complexity and measures of these features. There are several purposes for this. First, the measures provide a tool-independent estimate of the difficulty of a testcase. This allows us to estimate the workload a testcase may incur on any given tool, e.g., trivial, impossible, or somewhere in between. Second, they provide dimensions along which to classify the testcases. Third, a user can apply the measures to a new testcase and, inferring from the benchmark results for a variety of tools, choose a tool which best fits the new testcase. In other words, the measures can be used to classify *tools* according to their performance profiles across the various dimensions.

**Constructing testcases.** This stage constructs a suite of testcases from a collection of real programs which are known to contain buffer overflow vulnerabilities. For each program we produce a series of testcases of gradually increasing complexity. The construction process aims to overcome the issues we identified in the initial assessment, specifically: (1) tool failure due to program syntax, (2) tool failure or timeout due to program complexity, and (3) polarized tool performance.

Furthermore, the set of testcases serves as a basis for measuring the improvement gained by a new optimization, technique, or bugfix. Likewise, providing more than one testcase may reveal particular combinations of constructs which are particularly difficult (or easy) for a tool. This may inform the development of new techniques for handling these specific forms.

**Evaluation.** In this stage we apply the benchmark to one or more tools and collect results. First, this allows us to test the benchmark: to see how well it meets our requirements and to uncover any practical issues before release. Our initial evaluation prompted us to edit many of the testcases and to write scripts to automate benchmarking and result collection. Second, the evaluation provides information about the performance of the tools and uncovers bugs and bottlenecks.

**Evolution and maintenance.** This stage involves the long-term maintenance of the benchmark. It is only briefly described here, as a full treatment is beyond the scope of this work. It involves several parts:

- A publicly accessible repository of testcases and benchmark results.
- An API on which to build scripts for automating testing and data collection of a new tool. The scripts serve to specify the testing procedure and how to interpret the results for a given tool.
- A standard format for storing results. Establishing this entails agreeing upon performance measures which may be difficult to apply uniformly to different classes of

tools.

- Periodic updates of the suite. This includes adding new testcases and removing obsolete ones.

## 2.4 Static Measures

In this section we define static measures for estimating the cost of analyzing a testcase using CEGAR. The basis for the measures arises from an understanding of the predicate generation strategy used in SLAM-like SMCs. The measures are computed from a *dependency graph* which represents the semantic dependencies of a given vulnerability. We use CodeSurfer [11] to construct and measure the dependency graph for a given program.

CodeSurfer is a "program-understanding" tool which allows the user to navigate through source code in various representations, such as the control-flow graph and function call graph, and to examine the results of standard dataflow analyses such as reaching definitions and slicing. It also includes an API (in Scheme) which provides a programmatic interface to these structures and analyses.

### 2.4.1 Predicate Generation in CEGAR: Modeling Dependencies

The runtime complexity of CEGAR is dominated by predicate abstraction and model-checking. The cost of these steps is in turn exponential in the size of the predicate set. Thus, a useful static measure is one which can be used to estimate the size of the predicate set. First, let us review the CEGAR predicate generation process with the example shown in Figure 2.1(a), in which all variables are integers.

Assume the property being checked is the reachability of the statement labelled `ERROR`. Figure 2.1(b) shows an initial abstraction of the program with a potential counterexample

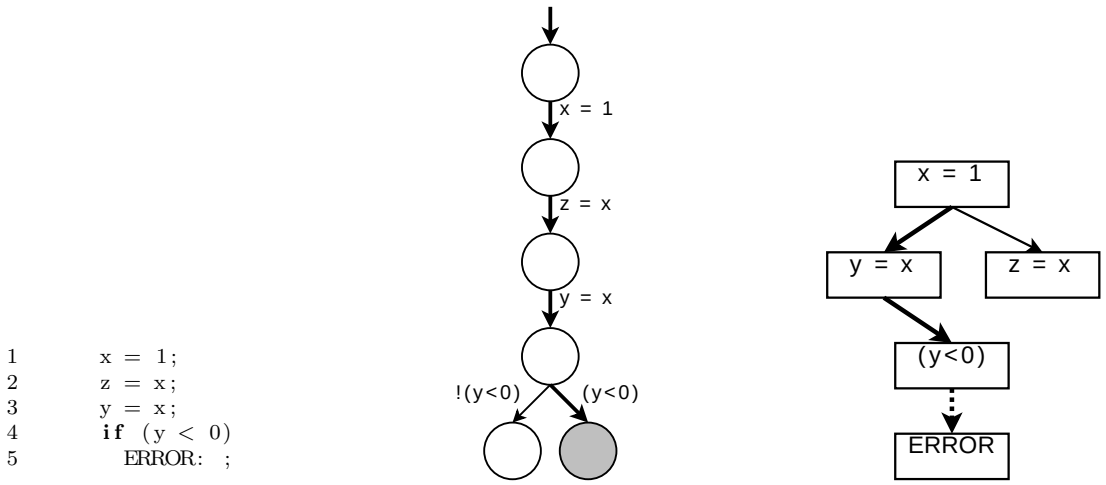



Figure 2.1: Example program (a), initial abstraction (b), dependency graph (c).

(cex) in bold and the error state shaded. It traverses lines 1 through 5, denoted $\langle 1, \ldots, 5 \rangle$. The cex is infeasible since it imposes the unsatisfiable constraint $x = 1 \wedge y = x \wedge y < 0$ and predicates are added to eliminate it from the subsequent model. Specifically, predicates are added to model two types of dependencies: data and control.

**Data dependency.** If a statement $t$ possibly uses a value defined in statement $s$, then $s$ is a data dependency of $t$. In the current example, `y = x` (definitely) uses the value defined by `x = 1`, so the latter statement is a dependency of the former. A data dependency is otherwise known as a *reaching definition*, a standard dataflow analysis [8].

Data dependencies of pointer expressions are defined by way of (may) *points-to sets*, which denote the variables which a pointer variable may point to at a particular program location. There are standard techniques for computing points-to sets such as [50]. In the presence of pointers, a pointer variable $p$ possibly uses the value of variable $v$ if the points-to sets of $p$ and $v$ overlap, i.e., if their intersection is non-empty. If $v$ is a non-pointer variable, its points-to set contains only itself: $\{v\}$.

**Control dependency.** For a conditional statement $s$ and a statement $t$, if $s$ guards the execution of $t$, i.e., if $t$ is executed iff $s$ evaluates to true, then $s$ is a control dependency of $t$. In the example above, `if (y < 0)` is a control dependency of the `ERROR` statement.

```
procedure PredGen(Cex c, Stmt s)
    S := {conditions of control dependencies in c}
    loop
        for each data dependency d ∈ c do
            for each predicate p ∈ S do
                S := S ∪ wp(d, p)
            end for
        end for
    end loop
    return S
end procedure
```

Figure 2.2: Predicate generation procedure.

Returning to the example in Figure 2.1(a), assume that a SLAM-like SMC adds predicates $y < 0$ and $x < 0$ to the predicate set by simulating the cex $\langle 1, \ldots, 5 \rangle$. Say predicate $y < 0$ is added first to model the control dependency between lines 4 and 5. Then, in a subsequent CEGAR iteration, $x < 0$ is added to model the data dependency between lines 3 and 4. Specifically, $x < 0$ is added since $x < 0 = wp(y := x, y < 0)$, i.e., since `y = x` defines a value used in `y < 0`.

Notice that even though line 1, `x = 1`, is a data dependency of line 2, `z = x`, no predicates are added to model this dependency since $wp(z := x, x < 0) = x < 0$ and similarly for $y < 0$. That is, `z = x` does not define a value used in any of the previously generated predicates. No new predicates are added for the data dependency between lines 1 and 3 since $wp(x := 1, x < 0) == 1 = 0 ==$ **false** and $wp(x := 1, y < 0) = y < 0$, which is already in the predicate set.

In general, to eliminate a spurious cex $c$ establishing the reachability of statement $s$, predicate generation follows the procedure shown in Figure 2.2. The procedure yields a predicate set composed of (1) conditions of control dependencies in $c$ and (2) a finite number of applications of $wp(d, p)$, where $p \in S$ and $d$ is a data dependency occurring in $c$. Notice that the bounds of the **loop** are unspecified. Determining the size of (1) is straightforward; we simply count the control dependencies appearing in $c$. However, we only estimate the size of (2), since computing it exactly may require work equivalent

to CEGAR. Specifically, we approximate the data dependencies to which *wp* is applied by using a *dependency graph*. We discuss the estimation of the *number* of applications of *wp* further below in Section 2.4.3 which covers programs with loops; for now we only consider programs without loops.

### 2.4.2 Dependency Graphs

A dependency graph is a representation of a program's dependencies constructed using the following rules:

- Add a node $s'$ for each statement $s$
- Add a directed edge $D(s', t')$ if $s$ is a data dependency of $t$
- Add a directed edge $C(s', t')$ if $s$ is a control dependency of $t$

A *combined dependency graph* (CDG) is a dependency graph where the edge labels are omitted, i.e., in which control and data dependencies are not distinguished. Assume there is a unique *target* which is the statement of interest. In the example in Figure 2.1(a), the target is the `ERROR` statement. Define a *trace* in a dependency graph to be any acyclic path from any node to the target node. Finally, define the *backward slice* as the subgraph of the CDG composed of all traces. This definition coincides with the standard one: the set of all statements that may influence (1) whether the target is executed and (2) the values of the variables used at the target [47].

The dependency graph for the example program is shown in Figure 2.1(c); data dependence edges are solid, control dependence edges are dotted. Consider the trace denoted in bold. Notice that the predicates generated by CEGAR to eliminate the cex, namely $y < 0$, $x < 0$, and $x > 0$, arise from the dependencies appearing in the trace and that the node for `z = x`, for which no predicates are generated, is not part of any trace. The trace illustrates a chain of transitive dependencies: the immediate control

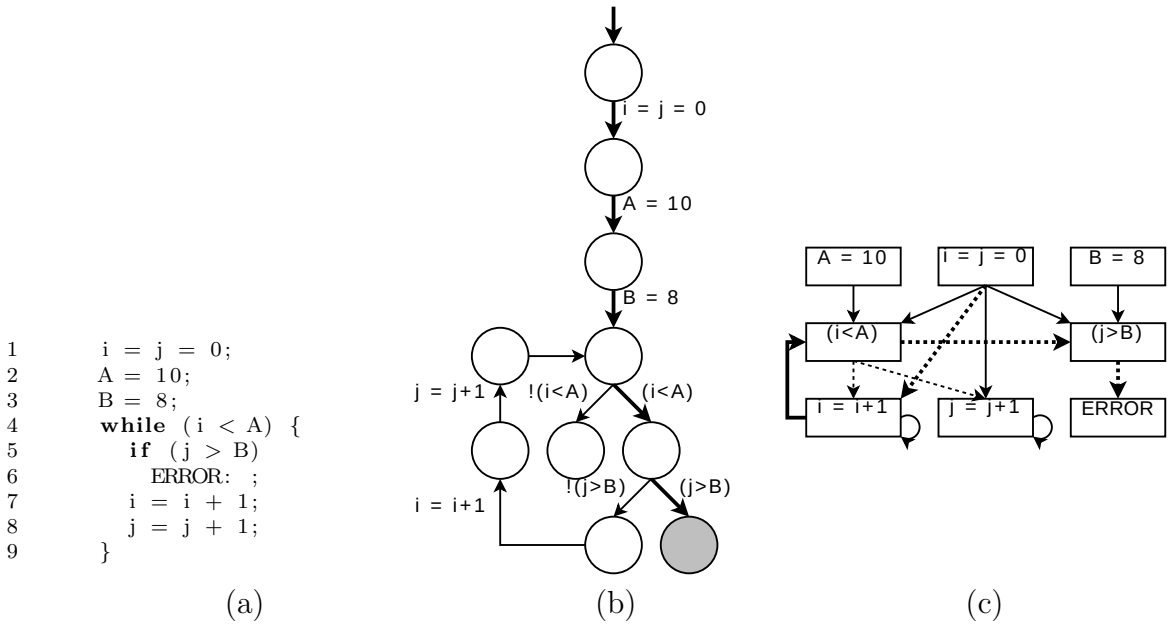



Figure 2.3: A program with a normal loop (a), its initial abstraction (b), and its dependency graph (c).

dependency of the target, `(y < 0)`, the subsequent data dependency `y = x` which defines the value of `y` used in `(y < 0)`, the next data dependency `x = 1` which may be used in `y = x`, and so on.

Comparing the trace in the example to the predicates generated to verify it, we conjecture that *the length of a trace is an approximation of the size of the predicate set generated by CEGAR to check the reachability of the target.*

### 2.4.3 Programs with Loops

We now consider the effect of loop constructs on CEGAR analysis complexity. Specifically, we consider what we call a *normal* loop which executes the loop body once for each element of a fixed-size array. For example, in the program in Figure 2.3(a), in which all variables are integers, say `A` and `B` denote the sizes of two arrays.

Consider how a SLAM-like SMC generates predicates in checking the reachability of the `ERROR` statement in this example. Initially, it adds predicates $i < A, j > B, i < 10, j > 9$ to eliminate the spurious cex in the initial abstraction shown in Figure 2.3(b).

Subsequently, it introduces predicates $i + 1 < A, j + 1 > B, i + 1 < 10, j + 1 > 9$, applying the loop of the predicate generation procedure once. This process continues until predicates $i + B + 1 < A, j + B + 1 > B, i + B + 1 < 10, i + B + 1 > 9$ are added, and the cex in the final model is valid.

From this simple example, we observe that the size of the final predicate set is roughly proportional to `A` which bounds the number of iterations of the `while` loop. Moreover, the final predicate set is obtained by roughly `A` applications of the predicate generation loop. The actual size of the predicate set is the size of the initial set of predicates (4) multiplied by `B+1` (9), but multiplying by `A` (10) gives us a reasonable upper bound.

Now consider the dependency graph for this example, shown in Figure 2.3(c). The graph shows the (transitive) dependency of the target on the `while` loop. The length of the trace in bold roughly corresponds to the size of the initial predicate set. From this example we make the conjecture: *the size of the predicate set generated for a target which depends on a single loop can be estimated as the product of a trace length and the loop bound.* In the case of nested (normal) loops, each of which iterates over a separate array, where the target is in the innermost loop, we estimate the size of the predicate set to be (trace length) $\times$ (product of the array sizes).

### 2.4.4 Computing Static Measures with CodeSurfer

In this section we describe the CodeSurfer API and the CodeSurfer plugin we implemented to compute measures of the dependency graph for a given program.

**CodeSurfer dependency graph API.** CodeSurfer defines a data-structure called the *system dependence graph* (SDG) which is similar to the dependency graph we defined. A subgraph of the SDG representing a single function is called a *procedure dependence graph* (PDG). The SDG differs from our definition in several respects.

A node in a SDG is called a *program point*. Like the nodes in our dependency graph, each statement is represented by a program point, but an SDG includes additional points

for declarations, the entry and exit points of a function, function call arguments (formal and actual), and a *result* variable capturing the value returned by a `return` statement. Traces in an SDG may be longer than those in the corresponding dependency graph due to these additional points.

Each program point and edge in the SDG has an attribute called its *kind*. The basic kind of program point is an *expression*, which represents an assignment statement. There are many kinds of program points, most of which do not represent visible program constructs. In our plugin, we filter program points by kind in order to avoid inflating the lengths of traces with artificial points. The two basic kinds of edges are control- and data-dependence edges, corresponding to the two types of edges in our dependency graph.

**CodeSurfer plugin.** The plugin takes as input an SDG, computed by CodeSurfer before the plugin is invoked, and a string. For each statement $s$ with a label containing the string, it computes a depth-first search of the SDG starting at $s$ and proceeding backwards to dependencies of $s$. The search along a branch terminates if a cycle or leaf is detected. The output is the lengths of all traces of $s$. Pseudo-code for the plugin is shown in Figure 2.4. The notation $\langle s \rangle$ is a path containing only the PDG vertex (program point) $s$, $\{p\}$ is the set of vertices in path $p$, $|p|$ is the length of path $p$, and $p \circ t$ is the path formed by appending vertex $t$ to path $p$.

Procedure `DFS` simply calls `DFSPath` with a path containing only the vertex $s$. Procedure `DFSPath` takes a path $p$ and recursively walks to each of the predecessors of its last vertex, which is returned by `Tail`. Procedure `Predecessors` returns the set of vertices with a (control or data) dependence edge targeting the given vertex $s$, i.e., the predecessors of $s$ in the SDG. The CodeSurfer API provides functions for accessing the predecessors of a given PDG vertex and for manipulating sets of PDG vertices.

For example, given the dependency graph shown in Figure 2.3(c) and the string `ERROR`, `DFS` calls `DFSPath`($\langle$`ERROR`$\rangle$). `Tail`($\langle$`ERROR`$\rangle$) returns `ERROR` (the program point

```
1: TL := ∅                                              ▷ Stores trace lengths
2: Input: a PDG vertex s
3: Output: a set of trace lengths
4: procedure DFS(PDGVertex s)
5:     DFSPath(⟨s⟩)
6:     return TL
7: end procedure
8: procedure DFSPath(Path p)
9:     t := Tail(p)
10:    P := Predecessors(t)
11:    P' := P − {p}                                    ▷ Remove cycles
12:    if P' is empty then                              ▷ Leaf detected
13:        TL := TL ∪ {|p|}
14:        return
15:    end if
16:    for each d ∈ P do                                ▷ DFS each predecessor
17:        DFSPath(p ∘ d)
18:    end for
19: end procedure
```

Figure 2.4: SDG DFS pseudo-code.

representing the `ERROR` statement). `Predecessors(ERROR)` returns the set $\{(\texttt{j} > \texttt{B})\}$, so `ERROR` is not a leaf, and `DFSPath` recursively calls itself with `DFSPath`$(\langle \texttt{ERROR}, (\texttt{j} > \texttt{b})\rangle)$. Now, `Tail`$(\langle \texttt{ERROR}, (\texttt{j} > \texttt{b})\rangle)$ returns `(j > b)` and `PreDecessors((j > b))` returns $\{(\texttt{i} < \texttt{A}), \texttt{i} = \texttt{j} = \texttt{0}, \texttt{B} = \texttt{8}\}$. Suppose the **for** loop selects `B = 8`: `DFSPath` recurses with `DFSPath`$(\langle \texttt{ERROR}, (\texttt{j} > \texttt{b}), \texttt{B} = \texttt{8}\rangle)$. This time, `PreDecessors(B = 8)` returns an empty set, so we add the length of the current path ($|\langle ERROR, (j > b), B = 8\rangle| = 3$) to $TL$. This continues until all program points have been explored; the final value of $TL$ is $\{3, 4, 5\}$.

The implementation has details not present in the pseudo-code and also allows for certain degrees of freedom:

- The kinds of vertices returned by `Predecessors` are configurable. The plugin currently filters out all but the basic kinds of vertices: expressions (assignments), control points, and function parameters.

- The number of branches (predecessors) per vertex and the total number of paths

explored by `DFSPath` is configurable. These parameters significantly affect the runtime of the plugin, since even a small program may have a relatively large SDG.

- If the target is a function call, e.g., `strcpy(dest,src)`, the program points for the arguments, i.e., `dest` and `src`, are passed to `DFS`, since the essential dependencies are those of the arguments, not the call site.

- Since a program can contain more than one target statement with a matching label, the output is grouped by target statement.

**Measures.** Given a set of trace lengths for an individual target statement, we can compute several basic measures using the plugin: maximum trace length, average trace length, and number of traces.

**Example.** Here we provide an example of the measures computed by the plugin for a series of related testcases taken from the Verisec suite, shown in Figure 2.5. In each of the cases, the target statement is labelled `VULN`. The code introduced in each successive case affects the reachability of the target statement, that is, the code changes the backwards slice of the target. The first case, `no_test`, performs no tests on the input data, the second, `med_test`, performs two tests, e.g., `fbuf[fb] == '\n'`, and the third, `heavy_test`, performs five tests. This increasing complexity is reflected in the analysis cost and static measures of each testcase, as shown in Table 2.1 (`DFS` was limited to 1000 traces and 10 branches per vertex in these trials). For example, in the second row of the table, which shows the result for the second test case, `med_test`, SatAbs generated 12 predicates to check the safety of the target statement labeled `BAD`; the CodeSurfer plugin found 484 traces for the target; the maximum trace length was 14 and the average trace length was 12.2.

Notice that the cost of checking the reachability of the target statement, measured in the size of the predicate set generated by SatAbs, increases with the PDG measures. This agrees with the intuition that (1) a trace represents a chain of transitive dependencies of

```
char fbuf[MAXLINE+1];
int fb;
int c1;

fb = 0;
while ((c1=nondet_int())!=EOF)
{
  fbuf[fb] = c1;
  fb++;
}
if (fb > 0)
{
  /* BAD */
  fbuf[fb] = EOS;
}
```

no_test

```
char fbuf[MAXLINE+1];
int fb;
int c1;

fb = 0;
while ((c1=nondet_int())!=EOF)
{
  if (c1 == '=')
    continue;

  fbuf[fb] = c1;

  if (fbuf[fb] == '\n') {
    fb--;
    if (fb < 0)
      fb = 0;
    else if (fbuf[fb] != '\r')
      fb++;

    fbuf[fb] = 0;
    fb = 0;
  }
  else
    fb++;
}

if (fb > 0) {
  /* BAD */
  fbuf[fb] = 0;
}
```

med_test

```
char fbuf[MAXLINE+1];
int fb;
int c1;

fb = 0;
while ((c1=nondet_int())!=EOF)
{
  if (isascii (c1) && isspace (c1))
    continue;
  if (c1 == '=')
    continue;

  fbuf[fb] = c1;

  if (fbuf[fb] == '\n')
  {
    fb--;
    if (fb < 0)
      fb = 0;
    else if (fbuf[fb] != '\r')
      fb++;

    fbuf[fb] = 0;
    fb = 0;
  }
  else
    fb++;
}

if (fb > 0)
{
  /* BAD */
  fbuf[fb] = 0;
}
```

heavy_test

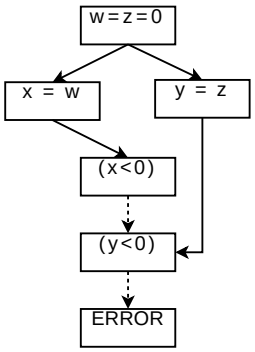
Figure 2.5: Three successively more complex testcases derived from Sendmail CVE-1999-0047.

| Testcase | #Preds | Max. Trace Len. | Avg. Trace Len. | # Traces |
|---|---|---|---|---|
| no_test | 0 | 7 | 6.5 | 14 |
| med_test | 12 | 14 | 12.2 | 484 |
| heavy_test | 18 | 24 | 21.3 | $\geq 1000$ |

Table 2.1: SatAbs analysis cost and trace length measures of three related testcases.

```
1  w = z = 0;
2  x = w;
3  y = z;
4  if (x < 0)
5     if (y < 0)
6        VULN: ;
```

(a)




(b)

Figure 2.6: A program with significant dependencies in separate traces (a), its dependency graph (b).

the target, (2) CEGAR generates predicates in order to model these linked dependencies, and (3) the cost of CEGAR analysis is determined by the size of the predicate set. We discuss the accuracy of the measures over a larger set of testcases in the evaluation section.

### 2.4.5 Threats to Measure Validity

There are several factors which may affect how accurately the PDG measures estimate the analysis cost of a given vulnerability.

**Multiple potential counterexamples.** In checking a vulnerability, CEGAR may generate predicates modeling dependencies occurring in more than one trace. In this case, measures of individual traces such as maximum or average length *underestimate* the analysis cost. A simple example in which this occurs is shown in Figure 2.6(a) along with its PDG in Figure 2.6(b). In this case, a CEGAR SMC may generate predicates $y < 0$ and $z < 0$, corresponding to the dependencies in the rightmost trace, but also generate predicates $x < 0$ and $w < 0$, corresponding to the leftmost trace. However, the individual traces do not reflect all the dependencies which are modeled by CEGAR. Some combination of the trace measures, e.g., a weighted sum of the lengths, may provide a more accurate estimate of the total size of the predicate set.

**Concise abstraction.** In some cases, a vulnerability may have many dependencies but CEGAR generates relatively few predicates in verifying it. That is, there may be dependencies occurring in the PDG which are ignored by CEGAR, so the PDG-based measures *overestimate* the analysis cost. This can occur if (1) a simple abstraction contains a valid cex, so no further predicate generation occurs, or (2) CEGAR generates a small set of predicates which eliminates all potential counterexamples. It is unclear how to detect when this is the case for a given program without replicating part of the potentially expensive CEGAR process.

**Conflation with simple measures.** It may be the case that CEGAR analysis cost has a stronger correlation with a simpler measure such as lines of code (LOC) and that the PDG measures are in fact indirect measures of LOC. Intuitively, this should not be the case, since one can artificially increase the LOC without introducing dependencies, e.g., by adding no-ops or irrelevant statements, and, consequently, without inducing the generation of additional predicates. Nonetheless, we check for the possibility of conflation with LOC in the evaluation section by comparing analysis cost with LOC.

## 2.5 Testcase Construction

In this section we describe the process by which we construct a series of testcases from a vulnerability. Recall that the basic objective of this process is to obtain a tractable testcase from real-world source code which cannot originally be analyzed by a given tool for technical reasons or resource limitations. The next objective is to produce, from a single vulnerability, a series of related testcases with varying complexity according to our dependency graph measures.

**Base case.** We first examine the source code for each vulnerability and, after understanding the reason for the error and the corresponding patch we identify the function in which the potential overflow occurs and slice away code outside its calling context. We

also identify the specific target statements which contain the vulnerable operations. We then parameterize all buffer size declarations by a preprocessor macro, `BASE_SZ`, so that we can control the bounds of buffer-dependent loops. At this point, we have reduced the original source code to a *base case*, from which we generate a series of testcases.

**Simplification.** Each testcase is obtained by applying a combination of simplifications to the base case. The simplifications eliminate or reduce (transitive) dependencies of the target statements, thereby reducing the analysis complexity. The simplifications are summarized in Table 2.2. Each row describes a single simplification: its type, an example of a statement before and after applying it, and whether it affects a data (D) or control (C) dependency. In most cases, it is clear why a simplification affects a particular kind of dependency. We briefly discuss some of the more subtle simplifications.

*Replace pointer with array.* Recall that the dependencies of a pointer expression consist of the dependencies of the variables in its points-to set. By replacing a pointer expression with a scalar or array expression, we reduce the points-to set to a singleton set. In applying this simplification, we consistently replace all expressions containing a particular pointer variable, introducing auxiliary variables as necessary.

*Inline function.* This removes the data dependencies in the implicit assignment between formal and actual parameters and between the function result and a result variable at the call site, e.g., `z` in `z = sum(x,y)`. This also removes the implicit control dependencies between the call site and function entry and exit points.

*Replace function with stub.* This substitutes a simpler function, a *stub*, for a more complex one. The simplest stub we use, `NONDET`, randomly returns 0 or non-zero. Using this stub enables a CEGAR SMC to model a potentially complex function as a single nondeterministic branch. We may also use a more complex stub which partially implements the original function. For example, the function `strstr` (substring search), could be replaced by `strchr` (single character search). Similarly, `strlen` (string length) could be replaced by a function which returns a random value in a given interval.

| Simplification | Before | After | D | C |
|---|---|---|---|---|
| Remove assignment | `x = y;` | — | X | |
| Replace pointer with array | `p++; *p;` | `i++; a[i];` | X | |
| Inline function | `z = sum(x,y);` | `z = x + y;` | X | X |
| Variable/constant propagation | `x = y; z = x;` | `z = y;` | X | |
| Remove branch/loop | `if (c) x = y;` | `x = y;` | X | X |
| Simplify branch condition | `if (c && d)` | `if (c)` | X | X |
| Replace function with stub | `if (strstr(s, t))` | `if (NONDET())` | X | X |

Table 2.2: Testcase simplifications

To choose which simplifications to apply to a given vulnerability, we first experiment with model checkers and base cases, identifying source code constructs which appear to incur the greatest analysis cost. We then select simplifications which affect these constructs. However, we restrict the simplifications to those that preserve the input language of the vulnerability. That is, the simplifications retain existing attack inputs, but may create new ones. For example, removing a branch which aborts on malformed input allows previously rejected input to produce an overflow.

The simplification process continues until the programs are reduced to a form which we believe current CEGAR SMCs can effectively handle. We then review the generated testcases and remove redundant and uninteresting ones. Finally, we apply the official source code patch, possibly modified for compatibility with our simplifications, to obtain a safe variant of each testcase. In general, each vulnerability required between one and four days for a single person to understand, slice, and process into testcases.

### 2.5.1 Example

Figure 2.7 shows the base case for the Apache CVE-2006-3747 vulnerability, annotated with the simplifications which were applied to it to produce a set of testcases for the benchmark. The target statement is labelled `VULN`. Function `escape_absolute_uri` takes as input a string, `uri`, specifying a Uniform Resource Indicator (URI), and an integer, `scheme`, specifying the length of the prefix of the input denoting its "scheme". If the

```
1  #define TOKEN_SZ (BASE_SZ + 1)
2  void escape_absolute_uri (char *uri, int scheme) {
3    int cp, c;
4    char *token[TOKEN_SZ];
5    if (scheme == 0 || strlen(uri) < scheme)    // #1 Remove branch and body.
6      return;
7    cp = scheme;                                // #2 Propagate value of scheme.
8    if (uri[cp-1] == '/') {                     // #3 Remove branch.
9      while (uri[cp] != EOS && uri[cp] != '/') // #4 Remove one or both conjuncts
10       ++cp;                                  //    or remove loop.
11     if (uri[cp] == EOS || uri[cp+1] == EOS)  // #5 Remove branch and body.
12       return;
13     ++cp; scheme = cp;                       // #6 Remove both assignments.
14     if (strncmp(uri, LDAP, LDAP_SZ) == 0) {  // #7 Replace strncmp with stub.
15       c = 0;
16       token[0] = uri;
17       while (uri[cp] != EOS && c < TOKEN_SZ) {
18         if (uri[cp] == '?') {
19           ++c;
20           VULN: token[c] = uri + cp + 1;
21           uri[cp] = EOS;
22         }
23         ++cp;
24       }
25     }
26   return;
27 }
28 int main () {
29   char uri [URI_SZ];
30   int scheme;
31   uri [URI_SZ-1] = EOS;
32   scheme = LDAP_SZ + 2;
33   escape_absolute_uri (uri, scheme);          // #8 Inline function call.
34   return 0;
35 }
```

Figure 2.7: Example testcase from Apache CVE-2006-3747.

URI is of type LDAP, lines 7–24 check the syntax of the URI and identify tokens in the input, storing the token offsets in the array `token`. However, the bounds check on line 16, `c < TOKEN_SZ`, is incorrect, since `c` can be incremented on line 18, and then used to index into array `token` on line 19, the target statement. That is, `c` can have value `TOKEN_SZ` on line 19, which is greater than the upper bound of array `token`, `TOKEN_SZ-1`. The patched version changes the check on line 16 to `c < TOKEN_SZ-1`.

Each of the simplifications preserve the intrinsic nature of the base case while allowing

additional inputs to trigger the overflow. In the base case, to trigger the overflow the input must begin with the string "`ldap://`" and followed by a minimum number of "`?`" characters, e.g., "`ldap://??`"—the specific number of "`?`" characters depends on the size of the target buffer, `token`. The simplifications are denoted by the comments on the right hand side of the source code in Figure 2.7. For example, line 8 checks for a "`/`" character after the "`ldap:`" prefix of the input. Simplification #3 removes this check, so in addition input "`ldap:??`" triggers the overflow. Line 14 checks if the input has the prefix "`ldap:`" using `strncmp`, the substring-search function of the C standard library. Simplification #7 removes this check, so that input "`//??`" triggers the overflow. Lines 15–23 comprise the operations which characterize the vulnerability, so they are left unmodified in all testcases: the `while` loop iterates over the input, "`uri`", copying a pointer into `token` each time the "`?`" character is encountered in `uri`.

### 2.5.2 Testcase Documentation

Each vulnerability is accompanied by the following documentation: a link to the original source code of the associated program, the file(s) in the original source code containing the vulnerability, the names of the source files of our testcases (listed in order of complexity), an explanation of how the vulnerability works and how the patch removes the vulnerability, and definitions of the simplifications used in each testcase. Documentation for the vulnerability in Figure 2.5 is shown in Figure 2.8.

## 2.6 Suite Composition

The benchmark is composed of testcases derived from a variety of buffer overflow vulnerabilities in open source programs, summarized in Table 2.3. We analyzed 22 vulnerabilities in 12 programs, producing 298 testcases (half of these are faulty versions and the other half are patched). Most of the vulnerabilities come from the Common Vulnerabilities

```
-= CVE-1999-0047 =-

Vulnerable version: Sendmail 8.8.3 and 8.8.4
File: sendmail/mime.c
Download from: http://www.sendmail.org/releases/historic.php
Domain: Server

_ Vulnerable Functions and Buffers _

Function mime7to8 reads four characters at a time from a file and copies each
character into a fixed sized buffer. The number of elements copied depends on
the length of the input, but the pointer into the dest buffer is reset if a
'\n' is encountered. A typo (fbuf >= &fbuf[X], which is always false, instead
of fbufp >= &fbuf[X]) prevents the copying loop from stopping early if the end
of the dest buffer is reached. The patched version fixes the typo.

_ Decomposed Programs _

Zitser's model program:

mime7to8/
  mime7to8_{arr,ptr}_{one,two,three}_char*_{no,med,heavy}_test_{bad,ok}.c

Variants arr and ptr use array indexing and pointer operations, respectively.
Variants one, two, and three read (and test) one, two, and three characters
from input on each iteration of the while loop. Variant no only checks whether
the input char is EOF; med also checks whether the input is '=', '\n', or '\r';
heavy also checks the input with isascii and isspace.

_ Notes _

This is Zitser's sendmail/s4, simplified. BASE_SZ was originally 50.
```

Figure 2.8: Testcase documentation for Sendmail CVE-1999-0047.

| Program | Domain | # Vulns | # Testcases |
|---|---|---|---|
| Apache | Server | 2 | 36 |
| edbrowse | App | 1 | 6 |
| gxine | App | 1 | 2 |
| LibGD | Library | 1 | 8 |
| MadWifi | Driver | 1 | 6 |
| NetBSD libc | Library | 1 | 24 |
| OpenSER | Server | 2 | 102 |
| Samba | Server | 1 | 4 |
| SpamAssassin | App | 1 | 2 |
| BIND | Server | 2 | 22 |
| WU-FTPD | Server | 3 | 24 |
| Sendmail | Server | 7 | 63 |

Table 2.3: Suite Composition

and Exposures (CVE) database [25] while the rest appear in prior publications [36, 53]. Different types of programs use buffers in different ways, so we selected programs from a variety of domains.

The benchmark directory structure is organized by program, vulnerability, and function – for example, `OpenSER/ CVE-2006-6749/parse_expression_list/` contains testcases from a vulnerability in the `parse_expression_list()` function of OpenSER, reported in vulnerability CVE-2006-6749. Each testcase's filename ends in `bad`, indicating a vulnerable case, or `ok`, indicating a patched case. Furthermore, each vulnerable statement in a testcase is preceded by the comment `BAD` (`OK` in the patched cases). Finally, all testcases `#include` a header file, `stubs.h`, in which standard macros, typedefs, and library functions are declared. Simple implementations of these functions are provided in `lib/stubs.c`. For example, Figure 2.9 shows the stub for the `strcpy` function, which copies the string pointed to by `src` into the buffer pointer to by `dest`, simply by iterating over each character in `src` until a null character is encountered. The header file also defines an important preprocessor macro, `BASE_SZ`, which specifies the base buffer size by which every testcase is parameterized.

```
1  char *strcpy (char *dest, const char *src)
2  {
3    int i;
4    char tmp;
5    for (i = 0; ; i++) {
6      tmp = src[i];
7      dest[i] = tmp;
8      if (src[i] == '\0')
9        break;
10   }
11   return dest;
12 }
```

Figure 2.9: Stub function for `strcpy`.

## 2.7 Evaluation

### 2.7.1 Objectives

In this section we apply the benchmark to a CEGAR SMC, namely SatAbs [24], in order to evaluate the quality of the benchmark with respect to our original objectives and requirements. Requirements R1, R2, and R3, i.e., realism, verification and falsification, and comprehensibility, are satisfied by the construction of the benchmark. To address R4 (solvability) we present solvability results below. For R5 (configurability) we present results showing the relationship between tool performance and (1) buffer size and (2) the simplifications we introduced into the testcases. The results also show that the benchmark obtains a wide range of performance from the test subject, thereby satisfying the basic objective of a benchmark.

### 2.7.2 Experimental Setup

We chose SatAbs as the test subject because it provides automatic instrumentation of potential buffer overflows and thorough handling of the C language, particularly arrays and pointer arithmetic which are heavily used in our testcases. We used version 1.6 with the default model checker, Cadence SMV, and with the iteration limit disabled. For the evaluation, we configured SatAbs to check all relevant buffer overflow assertions. The test platform was a 3.0 GHz Intel Xeon with 2 GB of RAM. All tests were run with a

| | Buffer size | | | |
|---|---|---|---|---|
| Result | 1 | 2 | 3 | 4 |
| Success | 167 (76%) | 155 (70%) | 150 (68%) | 145 (66%) |
| Crash | 24 (11%) | 26 (12%) | 28 (13%) | 31 (14%) |
| Incorrect | 18 (8%) | 18 (8%) | 18 (8%) | 17 (8%) |
| Timeout | 11 (5%) | 21 (10%) | 24 (11%) | 27 (12%) |

Table 2.4: Solvability results for safe testcases (220 claims)

| | Buffer size | | | |
|---|---|---|---|---|
| Result | 1 | 2 | 3 | 4 |
| Success | 101 (43%) | 94 (40%) | 86 (36%) | 79 (33%) |
| Crash | 37 (16%) | 37 (16%) | 40 (17%) | 42 (18%) |
| Incorrect | 63 (27%) | 61 (26%) | 59 (25%) | 56 (24%) |
| Timeout | 36 (15%) | 45 (19%) | 52 (22%) | 60 (25%) |

Table 2.5: Solvability results for unsafe testcases (237 claims)

half hour (1800s) timeout at buffer sizes 1 through 4.

### 2.7.3 Solvability Results

Given a testcase, SatAbs identifies a set of *claims*: statements to check for buffer overflows. In total, we ran SatAbs on 120 safe cases and 120 unsafe cases, for which SatAbs generated 220 and 237 claims, respectively. We observed that the solvability results are affected by the base buffer size. For example, for a testcase with two claims A and B, SatAbs may successfully check both claims at buffer size 1, but only succeed for claim A at buffer size 2. For claim B at buffer size 2, SatAbs may have crashed, produced an incorrect result, or timed out. As such, we present solvability results in terms of the number of claims successfully checked at each base buffer size.

Tables 2.4 and 2.5 show the solvability results for the safe and unsafe testcases, respectively. Figure 2.10 shows the result as a bargraph; each bar shows the result distribution for the buffer size indicated beneath the bar. Each row shows the number of claims which obtained a particular type of result at buffer sizes 1 through 4; the numbers in parentheses show the count as a percentage of the total number of claims.

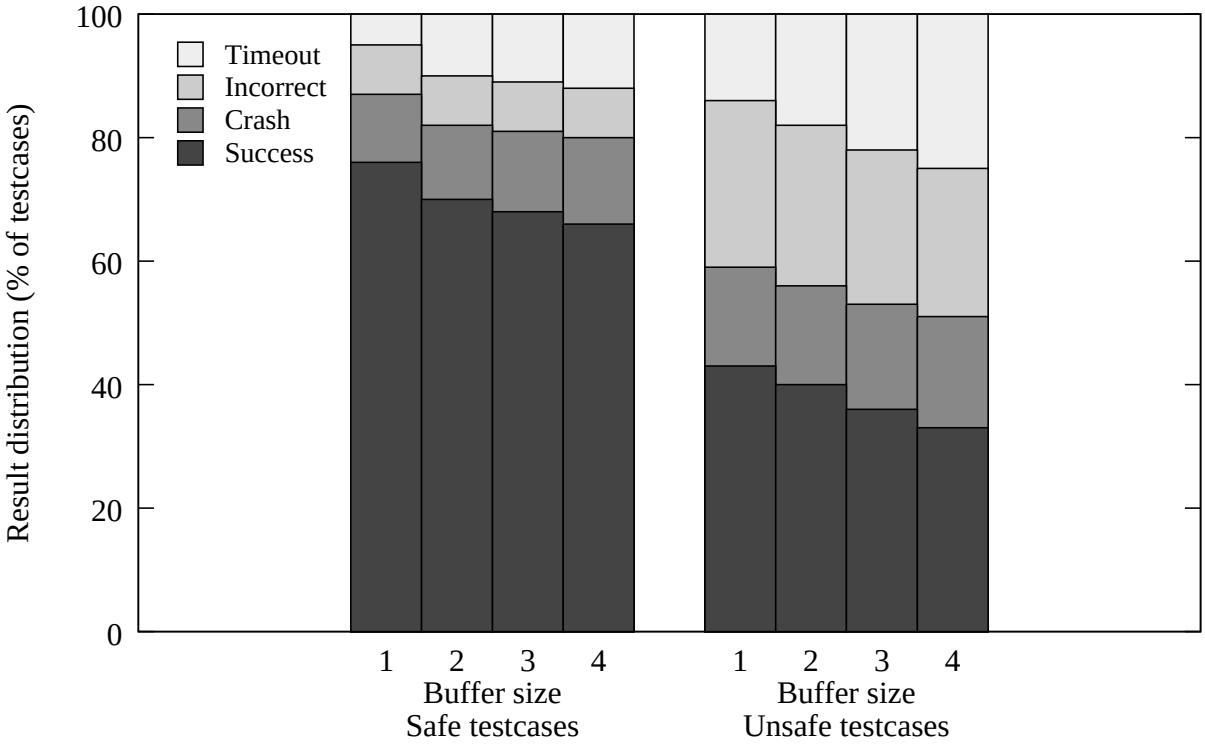



Figure 2.10: Solvability results as bargraphs.

In general, SatAbs' success rate falls between our rough bounds of 1/3 and 2/3. The success rate gradually decreases as the buffer size increases due to a corresponding increase in timeouts. Note that we also ran the benchmark at twice the timeout limit (3600s) with little change in the solvability results. In some cases, increasing the buffer size triggered bugs in SatAbs which caused it to crash or produce incorrect results. The timeout rates for the unsafe cases are from two to three times as high as those for the safe cases. One explanation for this is that in certain cases, the number of predicates generated is independent of the buffer size in the safe case, but grows with the buffer size in the unsafe counterpart. Thus, SatAbs succeeds on the safe case, but yields a timeout as the buffer size grows in the unsafe case. We discuss this behaviour in more detail below.

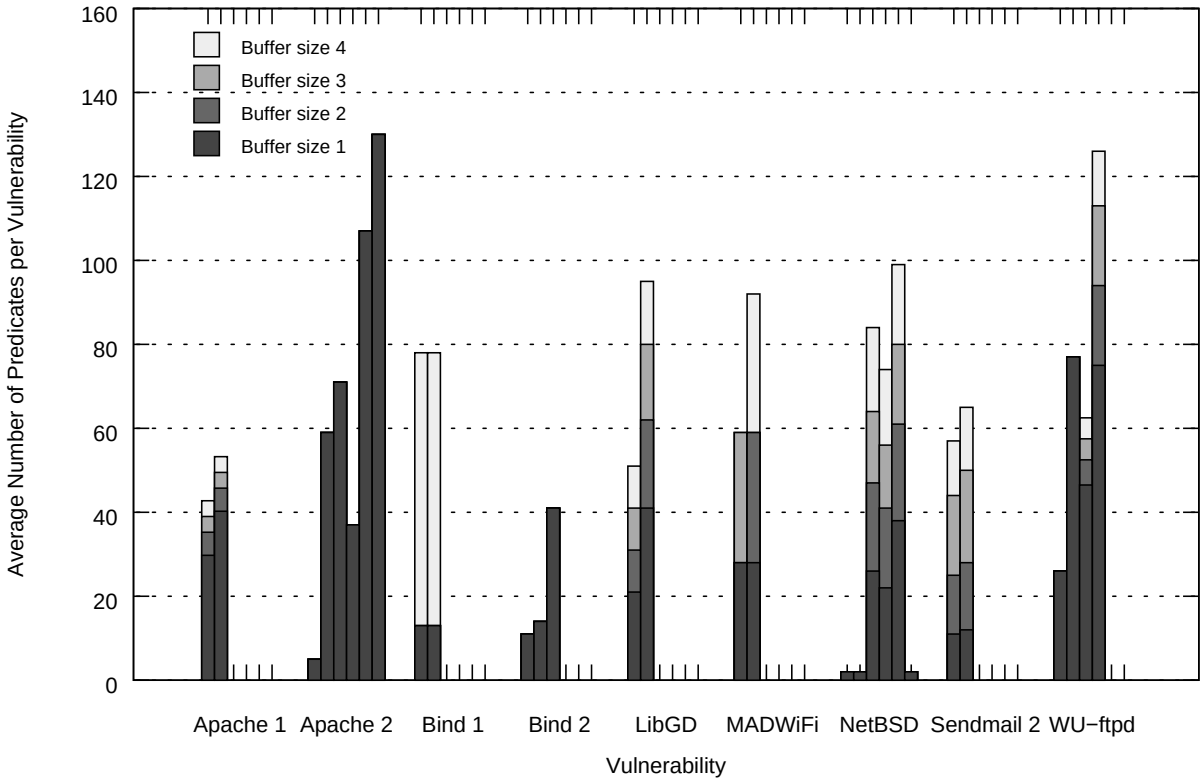



Figure 2.11: SatAbs performance results, safe cases.

### 2.7.4 Performance Results

In order to generate consistent performance results, we filtered the results so that only claims which were checked successfully at all buffer sizes were included. As such, some testcases were removed from the results altogether, since, for these, SatAbs failed to successfully check even a single claim at all buffer sizes. In sum, the performance results are collected over 90 safe and 90 unsafe cases with 136 and 76 claims, respectively.

Figures 2.11 and 2.12 show the performance results for the safe and unsafe vulnerabilities, respectively, with at most six testcases; results for OpenSER CVE-2006-6749 and Sendmail 1 CVE-1999-0047 are shown in Figures 2.13 and 2.14, respectively. Each bar indicates the difficulty of a testcase as the average number of predicates generated per claim (recall that a testcase can contain more than one claim). The results for buffer size 1 are shown by the darkest bar at the bottom, while results for buffer sizes 2 through 4

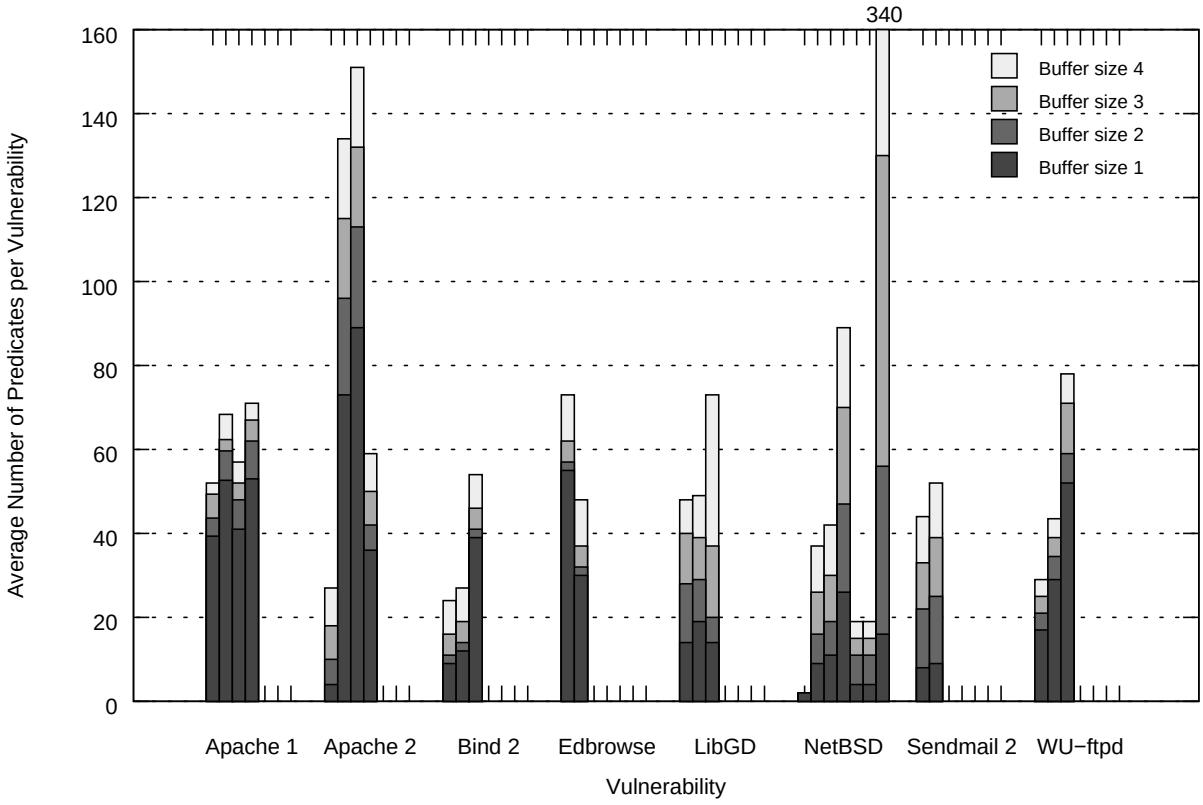



Figure 2.12: SatAbs performance results, unsafe cases.

are shown in order by the lighter bars stacked above. For example, the rightmost bar in Figure 2.11 indicates that the most complex of the testcases for WU-ftpd CVE-1999-0368 generated 75 predicates at buffer size 1 and 126 predicates at buffer size 4. The bars are clustered by vulnerability and sorted from left to right by order of testcase difficulty, that is, the leftmost testcase has the most simplifications and the rightmost has the fewest. In the case where bars do not appear for certain buffer sizes, the number of predicates generated remained the same as the buffer size increased. For example, the results at buffer sizes 2 through 4 for Apache 2 in Figure 2.11, are the same as those at buffer size 1.

The results in the first two figures confirm two trends which we expected to encounter, given our testcase construction methodology and understanding of CEGAR. First, for each vulnerability, the series of testcases produces gradually increasing analysis complexity. Also, the diversity of results shows that different vulnerabilities generate testcases

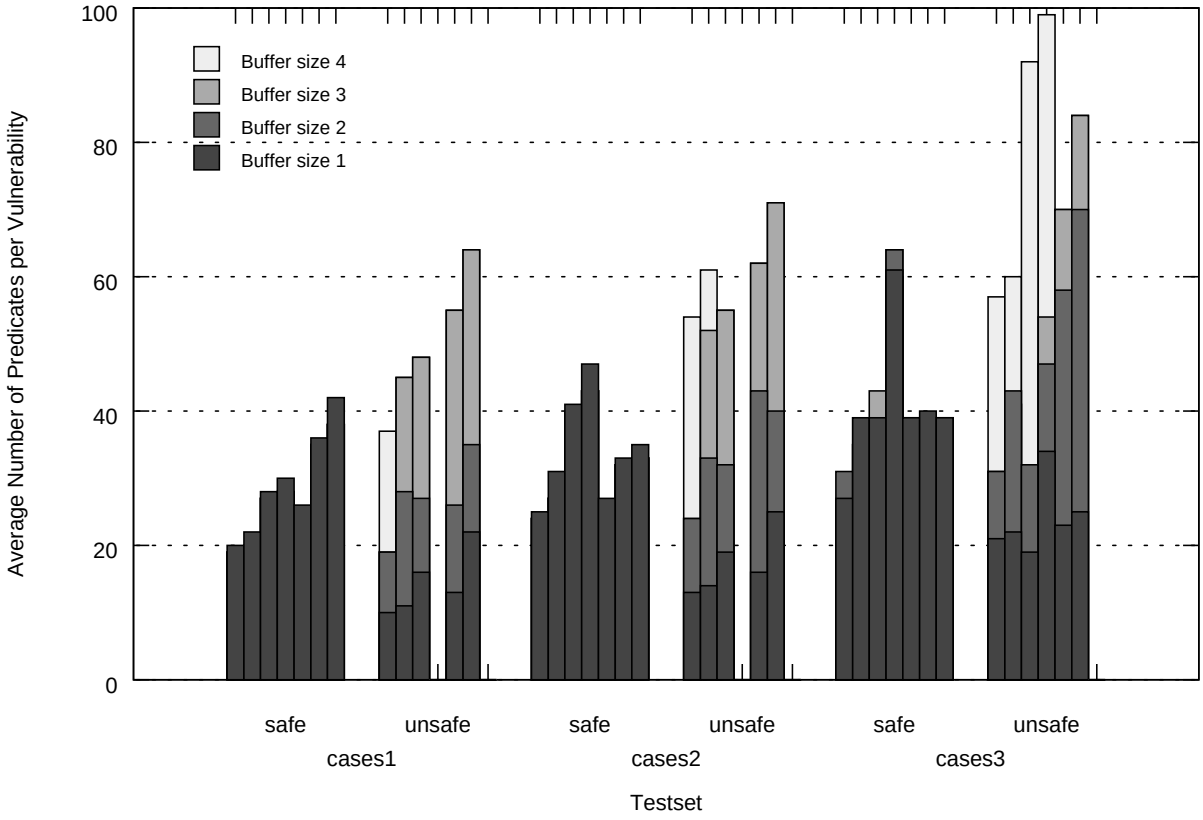



Figure 2.13: SatAbs performance results, OpenSER CVE-2006-6749.

with distinct difficulty bounds. For example, looking at Figure 2.11, we see that at buffer size 1, the most difficult case in Bind 2 (CVE-2001-0011) generates only 41 predicates, whereas the comparable case in Apache 2 (CVE-2006-3747) generates 130. Second, buffer-dependent loop bounds are shown in many cases to have a significant impact on analysis difficulty.

In some of the safe cases, such as Apache 2 (CVE-2006-3747), we see that SatAbs' performance is independent of buffer size; the number of predicates generated for these testcases remains the as the buffer size increases. The base source code for this vulnerability, before the patch is applied, is shown in Figure 2.7. Notice the conditional branch on line 17 guarding the vulnerable statement on line 20. The predicate appearing in the branch, `(c < TOKEN_SZ)`, which in the patched version is `(c < TOKEN_SZ - 1)` implies the safety of the array access, `token[c]`, on line 20. Thus, once CEGAR produces the

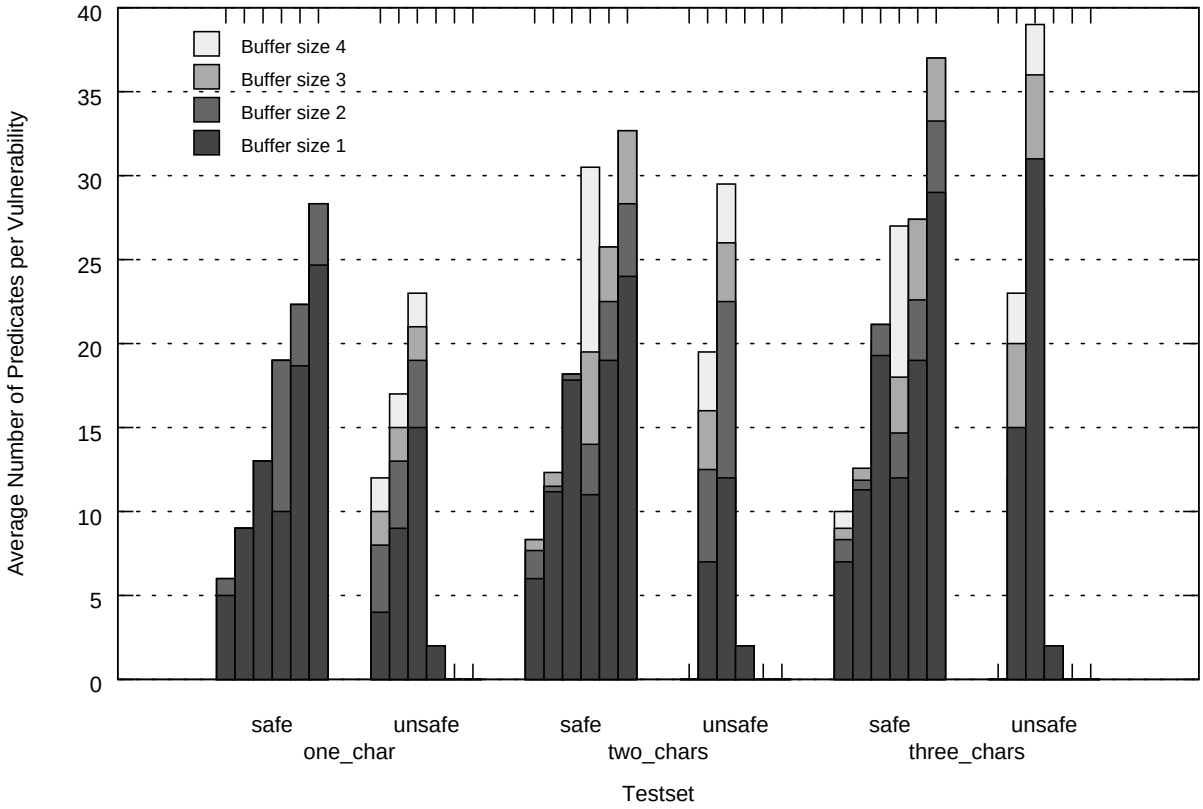



Figure 2.14: SatAbs performance results, Sendmail 1 CVE-1999-0047.

predicate `(c < TOKEN_SZ - 1)`, it is able to prove the safety of the array operation, regardless of the size of the array.

This trend only holds for safe testcases in which buffer operations are guarded by inequality checks on pointers or array indices; i.e. in which the check implies the safety of the buffer operation which it guards. In other testcases, for example those that use `strcpy()` to (safely) copy the contents of one array into a sufficiently large target array, the number of predicates needed to prove safety is roughly equal to those needed to find an error, and grows as a function of the target buffer size.

The bargraphs for OpenSER CVE-2006-6749 and Sendmail 1 CVE-1999-0047 show the results for these vulnerabilities for which we produced a more refined set of testcases. Note the change in the scale of the vertical axis from 160 to 100 for OpenSER and 40 for Sendmail, respectively.

The testcases for the OpenSER vulnerability are grouped according to a parameter called `cases`. The vulnerability involves looping over a string and checking whether each character matches one of several character constants. Testcases grouped under `cases1` check only whether the input contains a null character, those grouped under `cases2` check for both the comma and null characters, and those grouped under `cases3` check for the comma, null, and quotation mark characters.

Similarly, the Sendmail 1 vulnerability involves looping over a string and processing one or more characters in each iteration. The testcases are grouped according to the number of characters which are processed per iteration. For example, `one_char` testcases process one character per iteration. Within each of the groups, the testcases are sorted according to additional parameters. In the case of Sendmail 1, the first three tests have all pointer expressions replaced with array expressions, whereas the last three retain the pointer expressions. Then, each of these groups of three tests are sorted according to the number of conditional branches they contain. These testcases were constructed to isolate the effects of specific types of simplifications. The results confirm that each simplification has a significant effect on performance. For example, the impact of replacing pointers with arrays is clearly shown by the results for the Sendmail 1 testcases.

### 2.7.5 Static Measure Results

We also ran the CodeSurfer plugin on each testcase to obtain trace measures. The plugin was limited to explore at most 10 branches per program point and 1000 paths per program. Branches were explored in the arbitrary order selected by the `pdg-vertex-set-traverse` function of the CodeSurfer API. Since a testcase may have more than one target, i.e., more than one vulnerable statement, we collect the lengths of the longest traces for each target and take the average—the *average maximum trace length*.

The relationship between average maximum trace length and testcase difficulty, measured as the average number of predicates generated by SatAbs, is shown in the four

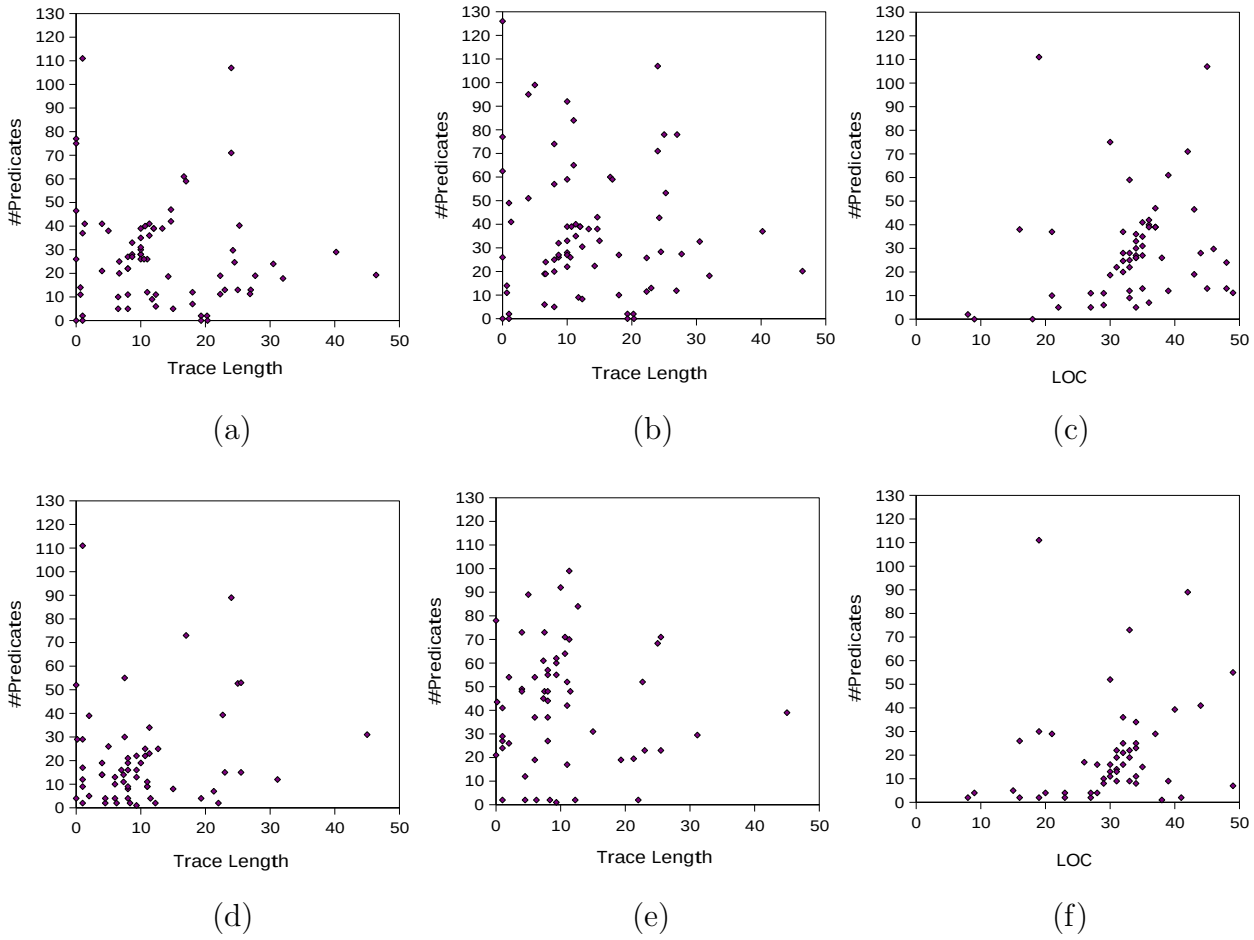



Figure 2.15: Comparison of relationship between static measures and testcase difficulty. The top row, (a)–(c), shows results for the safe cases, the bottom row, (d)–(f), shows results for the unsafe cases. Graphs (a) and (d) show the trace length measurements at buffer size 1 and (b) and (e) are for buffer size 4. Graphs (c) and (f) show the LOC measurements.

leftmost scatter plots on Figure 2.15. Each point denotes the result for a single testcase. The two plots on the right show the relationship between the number of lines of code (LOC) and testcase difficulty. Data points with LOC exceeding 50 or number of predicates exceeding 130 are omitted from the graphs.

The results show a positive correlation between both static measures and testcase difficulty. However, it is unclear to what degree trace length is a stronger measure of testcase difficulty than LOC. For example, with the safe cases at buffer size 1, the graphs for trace length and LOC look somewhat similar: the points are distributed along an

upwards slope (denoting a positive correlation) and form clusters, i.e., around 10 program points in the trace length graph and 35 LOC in the LOC graph. At buffer size 4, the clusters in the trace length graphs are dispersed vertically because of buffer-dependencies in the testcases, but retain the same positions in the horizontal axis.

Moreover, the trace length graphs at buffer size 1 look like the corresponding LOC graphs shifted to the left by about 30 units. This may be due to our testcase construction method which removes superfluous statements from the base code so that most remaining statements are significant, i.e. ones which lie in a trace. We may be able to obtain a more significant difference between the measures if we used larger, unprocessed testcases, e.g., testcases which measure high in LOC but low in trace length. Also, since the majority of our testcases measure less than 15 program points in trace length, we expect most of the results to be clustered around that value in the trace length graphs. To obtain more conclusive results, we need to experiment with testcases with a wider range of trace lengths.

## 2.8 Related Work

### 2.8.1 Buffer Overflow Benchmarks

Zitser et al. [53] constructed a benchmark suite from real C programs containing buffer overflow vulnerabilities. The benchmark was designed to evaluate a class of tools based on dataflow analysis. This type of analysis computes a program abstraction over a fixed abstract domain and thus captures only pre-specified facts about a given program's behaviour. As such, the analysis may fail to detect a vulnerability or produce a false alarm. The benchmark measures tool performance in terms of detection and false alarm rates. To enable this, the suite includes a patched (i.e., safe) counterpart of each vulnerable testcase. We adopted this methodology in the construction of the Verisec benchmark. The testcases in the Zitser benchmark are slices of real-world code and lack any further

| Name | R1 | R2 | R3 | R4 | | R5 |
|---|---|---|---|---|---|---|
| | | | | Detection Rate | False Alarm Rate | |
| Zitser | Y | Y | N | 40% | 20% | N |
| Kratkiewicz | N | Y | Y | 100% | 0% | Y |
| Wilander | N | Y | Y | 100% | 0% | N |

Table 2.6: Buffer overflow benchmarks evaluated under our requirements. R1: realism; R2: verification and falsification; R3: comprehensibility; R4: solvability; R5: configurability.

simplifications. As mentioned above, they obtained poor performance from the CEGAR SMC, Copper. Moreover, the suite only included one testcase per vulnerability and the testcases were not parameterized.

Kratkiewicz et al. [35] produced a benchmark consisting of small synthetic C programs with systematically varying syntactic constructs. The work is a follow-up to Zitser's, designed to evaluate the same class of tools, and uses the same methodology of including safe and unsafe testcases. The benchmark attempts to identify specific language constructs which account for a tool's false alarms and missed detections. However, the SMCs Copper and SatAbs obtained a perfect detection rate and false alarm rate of zero on the suite. Moreover, the tools which Kratkiewicz evaluated achieved a similarly high performance measure. This suggests that the parameters which the authors selected, i.e., the syntactic constructs, have a weak correlation with tool performance.

Similarly, the benchmark of Wilander et al. [51] is comprised of a set of yet smaller programs, each containing a single call to a standard C library function. The benchmark was designed to evaluate an even weaker class of tools based on syntactic analysis. These tools scan source code for specific syntactic patterns which may indicate a buffer overflow vulnerability and are thus highly susceptible to missed detections and false alarms. The SMC Copper obtained a perfect performance measure on this benchmark due to its specialized abstraction for C library functions.

An overview of the benchmarks described in this subsection is presented in Table 2.6. Each row of the table shows the evaluation of one benchmark with respect to our require-

ments. For example, the first row shows that the Zitser suite is realistic; includes both vulnerable and patched testcases; does not include comprehensible testcases; obtained a 40% detection rate (proportion of vulnerabilities found by Copper) and a 20% false alarm rate (proportion of safe cases deemed unsafe by Copper); and does not provide parameters for configuring the testcases.

### 2.8.2 Other Benchmarks

Atiya et al. present a benchmark for model-checkers of concurrent systems in [10]. The benchmark consists of a collection of problems, i.e. concurrent systems and their related properties, compiled from a survey of the model-checking literature. The authors defer the definition of performance measures to the "wider community." They also omit an evaluation of their suite, including instead the published results of case studies and experiments involving the sampled problems. The notable feature of this benchmark is that the authors classify the suite according to characteristics of the problems, such as *application domain*, *communication method*, and *property type*, and the types of techniques that have been used to solve them.

We initially attempted to create a similar classification scheme for the Verisec suite. Testcases were divided into groups by domain, e.g., *server*, *user application*, and *library*. We also defined two types of overflows: *length-based*, in which the overflow only depends on the length of the user input, and *content-based*, in which the overflow depends on the specific contents of the input. However, we did not find any relationship between CEGAR analysis complexity and domain or overflow type. Also, whereas Atiya et al. built their suite from testcases which were already documented with their classification features, we had to infer the overflow type from source code and bug reports, a time-consuming and unreliable process.

Dwyer et al. present a benchmark for evaluating explicit-state concurrent software model-checkers in [26]. The benchmark suite was compiled from a variety of sources and

includes both synthetic and real-world testcases which are seeded with errors. The work has two main objectives. The first is determining whether search order in an explicit-state model-checker (ESMC) has a more significant effect on performance than certain techniques do. The authors show that in some cases, simply randomizing search order yields a greater improvement over the default search order than a particular technique. The second objective is identifying program features which have a significant effect on ESMC performance; i.e., identifying program complexity measures. The authors find one such feature called *path error density* which is roughly the probability that a reference ESMC finds an error within 10000 runs using a random search order and a random scheduler. An interesting feature of their methodology is the use of a baseline analysis from which to evaluate a particular ESMC technique. Perhaps this approach could be adapted to the evaluation of CEGAR techniques. For example, we could use directed testing or bounded model-checking as a baseline analysis. In the case of directed testing, we may find a correlation between the size of the constraint set or the length of the error path and CEGAR performance; in the case of bounded model-checking, the size of the SAT instance or UNSAT core may be an indicator of CEGAR performance.

Pelanek presents an analysis of a benchmark for model-checkers of concurrent systems in [44]. The immediate objective of their work is to classify concurrent systems according to their structural and semantic properties. The authors hope to use the classification as a basis for automatically selecting the tool and the tool parameters which will solve a given problem most efficiently. The structural properties are computed from a given problem's state space graph, e.g., whether the graph is acyclic or consists of small connected components. The semantic classifications are obtained from a given problem's high-level description, e.g., the application domain from which the problem is derived. Unfortunately, the authors leave the evaluation of the benchmark to future work.

Singh et al. present a benchmark for multiprocessor (MP) systems in [52]. The benchmark consists of several parallel applications which, the authors claim, are representative

of the tasks which MP systems are typically used for, such as matrix factorization and physical simulations. The authors define several "behavioural characteristics" with which they classify the applications, e.g., the synchronization method used by an application. They also describe the syntactic structure of each of the applications' source code. The authors use these characteristics to (qualitatively) account for the performance profile of a test system across the benchmark. The work on testcase complexity measures in this thesis is loosely based on Singh's use of static characteristics to classify testcases.

Amla et al. present an evaluation of three bounded model-checking (BMC) techniques on a suite of industrial examples in [9]. They claim that counterexample depth, computed by an unbounded model-checker, is a significant factor in determining BMC performance. However, their results generally show a weak correlation between performance and counterexample depth, i.e., a wide variation in performance over a set of problems with a small variation in counterexample depth. On the other hand, the results do show distinctly different performance profiles for each of the BMC techniques. Unfortunately, they neglect to the discuss the features of the testcases which may account for these differences.

## 2.9 Conclusion

### 2.9.1 Limitations and Future Work

There are several aspects of the benchmark which can be improved upon in future work:

**Benchmark scope.** The vulnerabilities included in the benchmark were sampled non-randomly: we chose those which were interesting and which seemed to differ in form from previously selected ones. However, considering that CVE catalogs over 4000 buffer overflow vulnerabilities, our selection of 22 is very sparse, and it is likely that our sample excluded certain forms of overflows. Either a systematic or random sampling of a larger number of testcases would improve the realism of the benchmark, but this requires more

manpower than was available for this project.

**Testcase construction.** As discussed in the evaluation section, our testcase construction method yielded cases which were relatively small in both LOC and trace length measures. Because of this, the results obtained from the benchmark may not reflect the performance of tools such as SatAbs at a larger scale. However, recall that even with such small testcases SatAbs' success rate at buffer size 4 is only 33% for the unsafe cases.

Another significant limitation of our testcase construction method is that the simplifications were carried out by hand. We only generated fine-grained testsets for a few vulnerabilities, such as OpenSER CVE-2006-6749. Ideally, we would generate fine-grained testsets for all vulnerabilities, so that we could observe the effect of each simplification in isolation. However, this would require automation of the simplification process and it is unclear how this could be accomplished. This is because our testcase construction method requires that the simplifications preserve attack inputs, which in turn requires an understanding of the semantics of the testcase. As such, simple, syntax-based transformations would not be adequate for this purpose. One possible direction for supporting automatic testcase construction is static analysis for determining the attack language and its relationship to the source code.

SatAbs was used as the only reference tool in the testcase construction process This is problematic since the testcases were constructed around a single tool and may not obtain interesting or useful results when applied to another tool. However, given our time constraints, it would have been impractical to use more than one reference tool to construct the testcases, since each tool would have its own bugs and parser deficiencies to work around.

**Evaluation.** To improve readability, we reported the evaluation results for a given testcase as the average number of predicates generated by the test subject (in our case, SatAbs) for all buffer overflow claims over all buffer sizes between 1 and 4 (inclusive). However, for more than half the claims, there was at least one buffer size at which SatAbs

| Buffer size | Claim | | Result (average) |
|---|---|---|---|
| | A | B | |
| 1 | 5 | 10 | 7.5 |
| 2 | 7 | — | 7 |

Table 2.7: Example performance results.

failed to terminate successfully. Thus, the results for these claims were not included in the averages, as this would have misrepresented SatAbs' performance on the benchmark. To see why, consider Table 2.7 which shows the performance results for a fictional testcase with two claims, A and B: at buffer size 1 the test subject generated 5 and 10 predicates for each claim respectively; at buffer size 2 it generated 7 predicates for claim A and failed to check claim B. The average results are shown in the rightmost column. If we retain both claims in the results, it would appear that the tool's performance *improved* as the buffer size increased, yet it actually performed worse in claim A. Thus, for this testcase we would discard the results for claim B.

### 2.9.2 Lessons Learned

*Collaborate with the target community.* Sim defines a theory of benchmarking [46] in which (1) testcases are selected by a community of researchers and practitioners and (2) the benchmark is initially evaluated by tool developers. That is, the benchmark developers serve only to *facilitate* the development of the benchmark and its construction and evaluation are left to the community. In our case, we, the benchmark developers, performed both (1) and (2).

By following Sim's methodology we may have avoided the issue of limited benchmark scope, since the community would select all the testcases which it deemed relevant. The issues we encountered in testcase construction and evaluation might also have been avoided if we worked in conjunction with tool developers: testcase simplifications made to avoid parser or analysis bugs would be unnecessary and more complete results could be obtained. Moreover, testcases could be constructed independently of a reference tool,

since we would work with the developers to debug their tools until they succeeded on the testcases. Nonetheless, our testcase analysis and documentation provide an understanding of a variety of vulnerabilities and enable tool developers to concentrate on their tools, rather than the testcase code.

*Program structure affects CEGAR analysis complexity.* The benchmark provided insight into the difficulty of the buffer overflow problem and its relationship to CEGAR. Specifically, we identified two basic types of dependencies which account for the complexity of CEGAR analysis. We also discovered a form of overflow which is buffer-size-independent and therefore amenable to CEGAR analysis; moreover, we observed this form in several real-world cases.

* * *

The benchmark is available at `http://www.cs.toronto.edu/~kelvin/benchmark`. We encourage all SMC developers and users to download and experiment with the testcases and provide feedback.

# Chapter 3

# Supporting Buffer Overflow Analysis in YASM

YASM is a software model-checker for C. This chapter describes several changes made to YASM to support buffer overflow analysis in real-world programs. The chapter begins with an overview of YASM's architecture in Section 3.1, followed by a background on the techniques and data structures used in YASM in Section 3.2, an outline of the changes that were made to support buffer overflow analysis in Section 3.3 and detailed description of the changes in Sections 3.4 (theorem-prover interface), 3.5 (XML front-end), and 3.6 (predicate abstraction of pointer expressions). The chapter concludes with related work in Section 3.7 and a discussion of the current status of YASM and future work in Section 3.8.

## 3.1 YASM Architecture

YASM implements the standard symbolic software model-checking loop which combines predicate abstraction, model-checking, and counterexample-guided abstraction-refinement (CEGAR). The input is the source code of a C program and a temporal property to be checked on the program. The output is a truth value indicating whether the program satisfies the property:

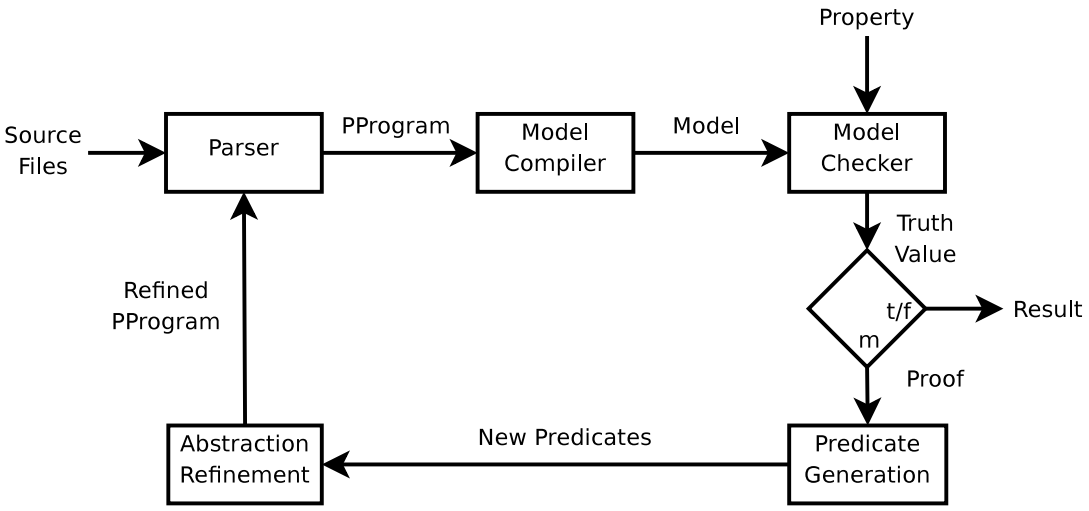



Figure 3.1: YASM architecture.

- True: the program satisfies the property
- False: the program violates the property
- Maybe: the analysis was inconclusive.

Figure 3.1 illustrates the YASM architecture. Components are denoted by rectangles, edges denote data-flow between components and are annotated with the type of data being passed, and the diamond denotes a conditional branch. The components are further discussed below.

1. **Parser.** The source code is parsed into an abstract syntax tree (AST). An initial predicate abstraction is computed from the AST by representing all statements as *skip* (no-op) and conditional branches as $*$ (non-deterministic choice). We call this a *nullary* or *trivial* abstraction since it is computed with an empty predicate set. The abstraction is represented as a predicate program (`PProgram`). Predicate abstraction and the `PProgram` data structure are discussed in Section 3.2.

2. **Model Compiler and Model Checker.** The `PProgram` is compiled into a finite-state model, namely, a Kripke structure [20], according to the semantics defined

(a)

```
1  void main (void) {
2    int x = 2;
3    int y = 2;
4    while (y <= 2)
5      { y = y - 1; }
6    if (x == 2)
7      { P1: }
8  }
```

(b)

```
1  void main (void) {
2    while (*)
3      { }
4    if (*)
5      { P1: }
6  }
```

(c)

```
1  void main (void) {
2    (x=2) := T;
3    while (*)
4      { (x=2) := (x=2); }
5    if (x=2)
6      { P1: }
```

(d)

```
1  void main (void) {
2    (x=2) := T;
3    (y <= 2) := T;
4    while (y <= 2)
5      { (y <= 2) := (y <= 2) ? T : *;
6        (x=2) := (x=2); }
7    if (x=2)
8      { P1: }
```

Figure 3.2: A simple C program (a), its initial abstraction (b), after adding $x = 2$ (c), after adding $y \leq 2$ (d).

in [30]. The model is checked for the property using the model-checker XChek [19]. If the property has a definite value in the model, namely, *true* or *false*, the process terminates. Otherwise, the model is too coarse to yield a conclusive result. The model-checker produces a proof which demonstrates why this is the case.

3. **Predicate Generation.** The abstraction is refined by adding predicates to the predicate set, following the abstraction-refinement strategy of [30]. The proof produced by the model-checker is traversed until an inconclusive transition is found. New predicates are obtained from the corresponding statement in the `PProgram`.

4. **Abstraction Refinement.** The predicate abstraction is re-computed using the updated predicate set. A theorem-prover, CVC Lite, is used to compute an approximation of the abstraction. The refined abstraction is checked by returning to step 2.

**Example.** The following example illustrates CEGAR analysis of a reachability property and is adapted from an example in [30]. Consider the program in Figure 3.1(a) and suppose we are checking for the reachability of the line labelled `P1`. This can be expressed as the CTL property $EF(pc = P1)$ [20]. Figures 3.1(b)–(d) show the three predicate pro-

grams which are constructed by CEGAR to check this property. Figure 3.1(b) shows the initial abstraction in which conditional branches are replaced with "$*$", denoting a non-deterministic choice, and all statements are omitted, indicating no-ops. Compiling and model-checking this `PProgram` yields value *maybe* since the reachability of the line labeled `P1` depends on the values of the non-deterministic branches. Specifically, there *may be* a path which exits the `while` loop and enters the `if` statement, but the feasibility of this path depends on the branch conditions. In particular, a predicate $x = 2$ is needed to determine whether the `if` statement is entered. Figure 3.1(c) shows the subsequent abstraction with the new predicate: $x = 2$ is initialized to **true** to model the statement `x = 2`, is unaffected by the body of the `while` loop, and is checked in the branch condition of the `if` statement. Compiling and model-checking shows that the line labeled `P1` is reachable if the `while` loop terminates, so the branch condition of the loop, $y \leq 2$, is added to the predicate set. The final abstraction is shown in Figure 3.1(d). The change of interest is the abstraction of statement `y = y - 1` which reads as follows: if $y \leq 2$ is **true**, then it remains **true** after decrementing `y`, otherwise its value is unknown ($*$); the value of $x = 2$ is unaffected. This abstraction is sufficient to determine that the loop does not terminate and thus the property $EF(pc = P1)$ is false.

## 3.2 Background

This section defines terms which are used in the rest of the chapter, specifically those related to parts of YASM which were modified.

**Predicate abstraction.** In order to apply model-checking to check a property of a program, a finite representation of the program is required. *Predicate abstraction* is a technique for constructing a finite representation of a program with an infinite data domain, first applied to C programs in [13]. A concrete state, i.e., a valuation of all program variables, is mapped to an abstract state according to its evaluation under a

finite set of *predicates*, i.e., boolean expressions over program variables and constants.

To see why even a trivial program can be an infinite state system, consider a statement in a program with a single integer variable $x$ as a binary relation $R$ over integers where $(y, y') \in R$ iff executing the statement with $x = y$ obtains $x = y'$. Then an assignment statement $s = x := x + 1$ is an infinite relation,

$$\{\ldots, (-1, 0), (0, 1), (1, 2), \ldots\}.$$

Suppose we abstract states with respect to the predicate $p = (x > 0)$: $p$ denotes the set of states which satisfy $p$, $\{s | s \models p\}$, and similarly for $\neg p = (x \leq 0)$. We represent statement $s$ as a finite relation over abstract states by mapping each state in $R$ to its evaluation under $p$, obtaining

$$\{(x \leq 0, x \leq 0), (x \leq 0, x > 0), (x > 0, x > 0)\}.$$

Below we show how we obtain this abstraction, but first we briefly define the data structure we use to store the abstraction.

**Predicate program.** The following definitions are adapted from [30]. First, define a *program* as a *control flow graph* (CFG): a directed graph where each edge is labeled with an *operation*. Let $V$ be the set of program variables. An operation is either (1) an assignment $v := e$, where $v \in V$ and $e$ is an expression over $V$ or (2) $assume(e)$ where $e$ is a boolean expression over $V$.

A *predicate program* is a CFG labeled with *boolean operations*. Let $P = \{p_1, \ldots, p_n\}$ be a set of quantifier-free boolean expressions over $V$. Then a boolean operation is either (1) a parallel assignment $p_1 := e_1, \ldots, p_n := e_n$ or (2) $assume(e)$, where the $e$ and $e_i$ are *partial boolean expressions* defined by the following grammar:

$$pbexpr ::= * \mid choice(boolexpr, boolexpr) \mid \neg pbexpr \mid boolexpr.$$

The symbol $*$ denotes an unknown expression and $choice(a, b)$ is defined as follows:

$$choice(a,b) = \begin{cases} \textbf{true} & \text{if } a \text{ is true} \\ \textbf{false} & \text{if } b \text{ is true} \\ * & \text{otherwise} \end{cases}$$

**Weakest precondition (WP).** Given a finite set of predicates, the goal is to compute a predicate abstraction of a program. We do this in a bottom-up fashion by computing the *weakest precondition* of a single predicate $p$ for a statement $s$. Intuitively, $wp(s, p)$ denotes those states in which executing $s$ makes $p$ true. The weakest precondition of $p$ wrt. an assignment $x := e$, written $wp(x := e, p)$, is $p[e/x]$: $p$ with $e$ substituted for all occurrences of $x$. For example,

$$\begin{aligned} & wp(x := x+1, x > 0) \\ &= (x > 0)[x+1/x] \\ &= x + 1 > 0 \\ &= x > -1. \end{aligned}$$

For the *assume* operation, $wp(assume(e), p)$ is simply $e \wedge p$.

**WP strengthening.** For a predicate set $P$ and a predicate $p \in P$, we may have $wp(s, p) = \phi \notin P$. In the previous example, with $P = \{x > 0\}$, we have

$$wp(x := x + 1, x > 0) = (x > -1) \notin P.$$

We need a finite abstraction, so we approximate $\phi$ by constructing a *strengthening* $\phi'$ over $P$ such that $\phi' \Rightarrow \phi$. Let $P = \{p_1, \ldots, p_k\}$. We define a *cube* over $P$ be a conjunction $c_1 \wedge \cdots \wedge c_k$ where $c_i \in \{p_i, \neg p_i\}$. We construct $\phi'$ to be the largest (weakest) disjunction of cubes such that $\phi' \Rightarrow \phi$, i.e., for any $\phi'' \Rightarrow \phi$, we have $\phi'' \Rightarrow \phi'$. Thus, $\phi'$ is the weakest precondition, over predicates in $P$, for $p$ to be true after executing $s$. To construct $\phi'$,

we use CVCL to check the implication $c \Rightarrow \phi$ for each cube $c$. Continuing the example, with $P = \{x > 0\}$, CVCL checks

$$\neg(x > 0) \nRightarrow (x > -1)$$

and

$$(x > 0) \Rightarrow (x > -1)$$

and returns $\phi' = (x > 0)$.

## 3.3 Overview of Changes

**Objective.** The overall goal of the work described in this chapter is to extend YASM to effectively check for buffer overflows in real-world C programs. The immediate goal is to provide the necessary data structures and algorithms for YASM to reason about pointers and arrays, as these are the essential constructs of a pointer-manipulating program. As a prerequisite, YASM must be able to parse all language constructs which appear in real-world code.

**Previous Limitations.** To understand the need for the changes made, we briefly review the limitations of the initial implementation of YASM, called OldYASM, below.

1. **Native theorem-prover interface.** YASM uses CVCL to compute WP approximations in the predicate abstraction step. CVCL is written in C++ while YASM is written in Java. In OldYASM, CVCL object code is linked into YASM by way of SWIG [7] which automatically generates Java Native Interface [5] (JNI) code to translate Java method calls and data to their C++ counterparts and vice versa. The SWIG templates are tedious to maintain, especially since the generated C++ code needs to be corrected by hand, and the opaque C++ exception interface makes it difficult to debug theorem-prover errors.

2. **ANTLR C Parser.** OldYASM uses the ANTLR parser-generator [1] and a grammar for GNU C [4] for the first stage of parsing. The grammar is incomplete and rejects certain language constructs which commonly appear in low-level code. Also, to reduce the number of language constructs handled in the rest of YASM, redundant constructs were removed from the grammar. A separate tool, CIL [41], is used to transform code to conform to the simplified grammar before passing it to ANTLR. However, like ANTLR, CIL also has incomplete language support and, furthermore, produces code which is difficult to read and relate back to the original code.

3. **Flat memory model.** In OldYASM, memory is modeled as a mapping from variables to integers. There is no notion of the *address* of a memory element. This model cannot faithfully represent an array, since this is a sequence of memory elements with contiguous addresses. Likewise, a pointer, i.e., a memory element containing an address, and pointer (address) arithmetic cannot be represented.

**Changes.** The changes are of two types: *improvements* to existing functionality and *extensions* which provide new functionality.

- Improvements

    1. **Java interface for a command-line theorem-prover.** The CVCL library is replaced by an interactive process which YASM communicates with via pipes. A class is added to translate from YASM's internal expression language to CVCL's presentation input language.

    2. **Ximple front-end.** The ANTLR parser is replaced by an XML parser which accepts Ximple programs. Ximple is an XML representation of GCC's internal representation of a C program, designed by Arie Gurfinkel. CIL is replaced by a modified version of GCC which produces Ximple output.

- Extensions

3. **Predicate abstraction of pointer expressions.** The predicate abstraction component is extended to handle expressions containing pointers and arrays. This is supported by a "logical memory model" in which memory is represented as a set of disjoint arrays. Classes are added added to translate between C expressions and CVCL expressions according to the new model.

## 3.4 Java Interface for a Command-Line Theorem-Prover

This section describes the design and implementation of an interface for a command-line theorem-prover. The interface communicates between YASM and an external theorem-prover process. It translates between YASM's internal expression language and the input language of the theorem-prover and forwards theorem-prover operations to the external process. The section begins with a background on the theorem-prover and the interface, followed by a description of the implementation details, and finishes with a brief discussion.

### 3.4.1 Background

**CVC Lite (CVCL).** A theorem-prover which is used by YASM to compute WP strengthenings. Specifically, CVCL is used to check the validity of an implication $\psi \Rightarrow \phi$ where $\psi$ and $\phi$ are quantifier-free formulas with integers and arrays. CVCL has a stack-based interface: each level of the stack contains one or more formulas called *assumptions*. This set of formulas $\Gamma$ is called a *logical context*. To issue a *query* of formula $\phi$ is to check the validity of $\bigwedge \Gamma \Rightarrow \phi$. If the implication is valid, CVCL returns the subset of $\Gamma$ which was used in the proof of validity. For example, if $\Gamma = \{x > 0, y > 0, z < 0\}$ and $\phi = x+y > 0$, then

querying $\phi$ under context $\Gamma$ is to check the validity of $x > 0 \wedge y > 0 \wedge z < 0 \Rightarrow x + y > 0$, which is valid. CVCL returns $\{x > 0, y > 0\}$ since $z < 0$ is not used in the proof of validity. The syntax for formulas accepted and output by CVCL is called the *Presentation Input Language* (PIL).

**CVCL Interface.** To efficiently compute a strengthening, YASM manipulates the logical context using a stack-based backtracking strategy in order to avoid redundant queries. As such, YASM defines an interface, `ITheoremProver`, for a stack-based theorem-prover with the following functions. YASM defines a class called `Expr` to represent abstract syntax trees (ASTs) and boolean expressions such as WPs and cubes. The `ITheoremProver` interface is summarized below.

Manipulating the stack:

- Function `push ()` increases the stack level by one, adding an empty level at the top.
- Function `pop ()` decreases the stack level by one, removing the top level.
- Function `stackLevel ()` returns the current level.

Modifying the current stack level:

- Function `declare (Expr n, Expr t)` declares identifier `n` to be of type `t`.
- Function `assertFormula (Expr f)` adds formula `f` as an assumption.

Querying:

- Function `query (Expr f)` checks the validity of `f` in the current context.
- Function `getAssumptionsUsed ()` returns the assumptions used in the previous (valid) query.

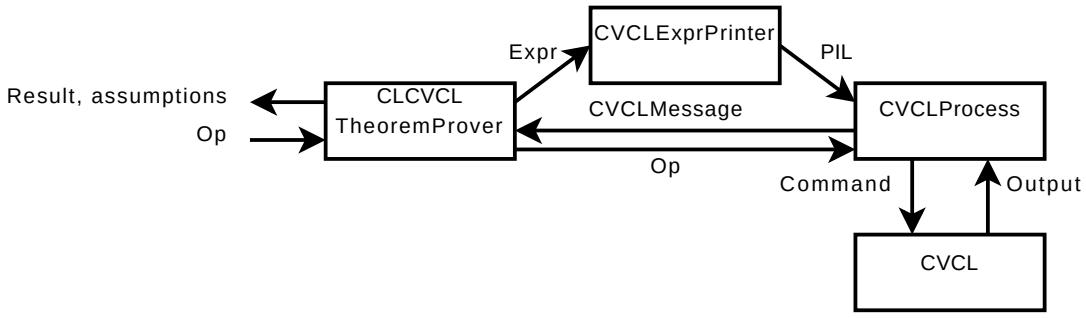



Figure 3.3: CVCL interface class layout.

### 3.4.2 Implementation

The two main issues in implementing the `ITheoremProver` interface on top of CVCL are

1. Communicating with the CVCL process

2. Translating formulas from `Expr` to PIL and vice versa.

Figure 3.3 shows the classes which implement the interface and the flow of data among them; rectangles denote classes and the edges are labeled with the type of data which is passed between classes.

**CVCLProcess.** This class initializes and manages a CVCL process and provides an interface for communicating with it. Calls to the interface are passed to the process as text. Output from the process is encapsulated in a CVCLMessage object with attributes for the elements of the output. The four types of CVCLMessage are summarized below:

- `VoidMessage`. Empty output with no attributes. Returned by `push`, `pop`, and `declare`.

- `AssertListMessage`. Contains a list of formulas as strings. Returned by `getAssumptionsUsed`.

- `ValidityMessage`. Contains a boolean which is true iff the previous query was valid. Returned by `query`.

- `WhereMessage`. Contains a list of formulas as strings and an integer representing the current stack level. Returned by `where`.

Command `where` is similar to `getAssumptionsUsed`, but it returns all assumptions in the current logical context, as well as the current stack level.

`CVCLProcess` raises an exception if the process produces unexpected output. It also prevents certain sequences of commands from being issued if they are invalid, since they would cause the process to abort or generate nonsense output. Specifically, it uses three variables to keep track of the query state:

- `boolean validQueryState`. True iff the previous `query`, if any, was valid.
- `boolean invalidQueryState`. True iff the previous `query`, if any, was invalid and the current scope level is greater than `invalidQueryStackLevel`.
- `int invalidQueryStackLevel`. The stack level of the previous invalid query; $-1$ if no invalid query has been made.

An exception is raised if any of the following commands is called in an invalid state:

- `where`, `assertFormula`, `push`, and `getAssumptionsUsed`, if `invalidQueryState` is true, since the context should not be examined or augmented if the query is invalid in the current context; in other words, the only legal operation after an invalid query is to reduce the context (by popping the stack).
- `getAssumptionsUsed`, if `validQueryState` is not true, since YASM only uses assumptions used to prove valid queries.

**CLCVCLTheoremProver.** This class implements the `ITheoremProver` interface on an underlying CVCL process by

1. Translating `Expr` formulas to PIL

2. Translating CVCL's text output to `Expr`

The first step is handled by `CVCLExprPrinter`, which traverses an `Expr` AST and emits PIL text as appropriate. As for the second step, the only problematic output of CVCL is the assumption list produced by `getAssumptionsUsed`. Translating these assumptions to `Expr` form is problematic since CVCL may print an assertion in a form that is logically equivalent, but syntactically different, from the original input. For example, a formula may have been asserted as $x > 0$ but is printed as $x \geq 1$.

However, there are three useful observations about the assumption format which enable us to avoid parsing the output of `getAssumptionsUsed`. First, the formulas returned by `getAssumptionsUsed` for a valid query are always a subset of the asserted formulas. Second, `where` returns all assumptions in the same order in which they were asserted, but possibly in a syntactically different form. Third, the assumptions returned by `getAssumptionsUsed` are a subset of those returned by `where` and, moreover, are syntactically equivalent. Thus, to recover the assumptions used to prove a valid query, we call `where` and `dumpAssumptions` and, for each assumption returned by the latter, we find its position in the output of `where`. We use this as an index into our list of Expr assumptions to recover the corresponding Expr.

### 3.4.3 Discussion

The CVCL interface required about three weeks for two students to implement. The majority of the time was spent debugging the part of `CVCLProcess` which parses the output from CVCL, which has several tricky corner cases. The output also changed subtly across releases of CVCL. Thus, deciding whether to implement a command-line theorem-prover interface significantly depends on how stable and consistent the output is. Class `CLCVCLTheoremProver` was straightforward to implement once we figured out how to map CVCL's assumption lists to the internal `Expr` list; our approach should be applicable to any theorem-prover with output that can be reliably parsed. Finally, class

`CVCLExprPrinter` was also easy to implement since both the `Expr` and PIL languages are simple and the mapping between them is clear. This should be the case with other theorem-provers, especially since our `Expr` language is limited to arithmetic expressions over integers and arrays.

## 3.5 Ximple Front-End

This section describes the design and implementation of a front-end for YASM which parses an XML representation of C (Ximple) and constructs a predicate program. It begins with a background on the Ximple file format and the data structures used to store intermediate representations of the input program in YASM. It then briefly describes the implementation of the front-end and concludes with a discussion.

### 3.5.1 Background

**Ximple.** Ximple is a type of XML document for representing C programs. It is based on an internal representation of GCC called Gimple [4]. Arie Gurfinkel has implemented a plugin for GCC which produces Ximple output for a C program. It produces two files: one defining datatypes and global declarations and another defining function bodies and local declarations. Figure 3.4 shows the Ximple document for a small C program. The major parts of a Ximple document are summarized below:

- Type declarations. All types are indicated with a `typeDecl` element; the `tuid` and `name` attributes specify a unique identifier and name for the type, respectively. A `typeDecl` for a built-in type, i.e., `long`, `int`, `short`, or `char`, contains a self-referring `typeRef`. Whereas a `typeDecl` for a user-defined typed, i.e., a `typedef`, contains a `typeRef` referring to the base type. In the example, `my_int` is a `typedef` for `int`, so the corresponding `typeDecl` contains a `typeRef` with the `tuid` for `int`, namely, 8.

```
1  typedef int my_int;
2  my_int global;
3  int fun (int n) {
4    int i;
5    if (i < 5)
6      i = i + 1;
7    return i;
8  }
```

(a)

```
1  <typeDecl tuid="8" name="int" uid="0">
2   <typeRef name="int" uid="0" tuid="8"/>
3  </typeDecl>
4  <typeDecl tuid="9" name="my_int" uid="1633">
5   <typeRef tuid="8"/>
6  </typeDecl>
7  <globals>
8   <varDecl  name="global" uid="1637">
9    <type>
10    <typeRef tuid="9"/>
11   </type>
12   <init>
13    <intcst value="0"/>
14   </init>
15  </varDecl>
16 </globals>
```

(b)

```
1  <functionDecl name="fun" uid="1650">
2   <resultDecl  uid="1651">
3    <type>
4     <typeRef tuid="8" name="int" uid="0"/>
5    </type>
6   </resultDecl>
7   <argList>
8    <parmDecl name="n" uid="1648">
9     <type>
10     <typeRef tuid="8" name="int" uid="0"/>
11    </type>
12   </parmDecl>
13  </argList>
14  <locals>
15   <varDecl  name="i" uid="1811">
16    <type>
17     <typeRef tuid="8" name="int" uid="0"/>
18    </type>
19   </varDecl>
20  </locals>
```

(c)

```
21   <body>
22    <bb id="0">
23     <stmtList>
24     <condExpr>
25      <lt>
26       <var name="i" uid="1811">
27       <intcst value="5"/>
28      </lt>
29      <goto>
30       <labelDecl labelUID="26" uid="1757"/>
31      </goto>
32      <goto>
33       <labelDecl labelUID="23" uid="1751"/>
34      </goto>
35     </condExpr>
36     </stmtList>
37     <succList>
38      <edge dst="1" trueValue="true" />
39      <edge dst="2" falseValue="true" />
40     </succList>
41    </bb>
42    <bb id="1">
43     <stmtList>
44     <labelExpr>
45      <labelDecl labelUID="26" uid="1757"/>
46     </labelExpr>
47     <modifyExpr>
48      <var name="i" uid="1811">
49      <plus>
50       <var name="i" uid="1811">
51       <intcst value="1"/>
52      </plus>
53     </modifyExpr>
54     </stmtList>
55     <succList>
56      <edge dst="2" trueValue="true" />
57     </succList>
58    </bb>
59    <bb id="2">
60     <stmtList>
61     <labelExpr>
62      <labelDecl labelUID="23" uid="1751"/>
63     </labelExpr>
64     <return>
65      <modifyExpr>
66       <result  uid="1651"/>
67       <var name="i" uid="1811">
68      </modifyExpr>
69     </return>
70     </stmtList>
71     <succList>
72     </succList>
73    </bb>
74   </body>
75 </functionDecl>
```

(d)

Figure 3.4: A simple program (a), its Ximple datatypes and global declarations (b), Ximple function bodies and local declarations (c), continued (d).

- Identifiers. Every identifier has a unique `uid`. The identifiers in the example are the names for types, variables, functions, function parameters, and the (anonymous) return variable for the `fun` function.

- Basic blocks. A function `body` is a list of basic blocks. Each basic block (`bb`) contains a list of statements (`stmtList`) and a list of control-flow successors (`succList`). A basic block has a single entry point, the first statement, and a single exit point, the last statement; only the last statement may be a branch. Ximple has two el-

ements for denoting a branch: a two-way branch, `condExpr`, corresponding to an `if` statement in C, and a multi-way branch, `switchExpr` (not shown in the figure), corresponding to a `switch`. Notice that the `condExpr` is the last (and only) statement in basic block 0. The successor list for basic block 0 indicates the basic blocks for the true and false targets of the branch. The successor list is redundant with the `goto` elements in the `condExpr`.

**CProgram.** This class is a representation of a C program. Its main parts are summarized below:

- CFG table. A map from function `uid`s to their corresponding control-flow graphs (CFGraph).
- `CFGraph`. Contains an array of `BasicBlocks` basic blocks for a given function indexed by `id`. Also provides a map from program labels to their corresponding basic blocks.
- `BasicBlock`. Contains a list of `Expr`s, one for each statement in the associated basic block, in syntactic order. Contains a list of control-flow predecessors and successors and pointers to the syntactic predecessor and successor blocks.
- `CSymbolTable`. Contains a map from `tuid`s to types and a map from `uid`s to variable declarations.

**Predicate Program Implementation (PProgram).** The structure of the `PProgram` constructed for a small program (Figure 3.5(a)) is shown in Figure 3.5(b). Each node in Figure 3.5(b) is label of a `PStmt`. The nodes in parentheses denote classes other than `PStmt`. Unless a line is labeled, a solid line denotes a syntactic pointer (the `next` attribute) and a dashed line denotes a control-flow pointer (the `dest` attribute). Labeled lines denote other types of attributes, e.g., the line labeled `functionDefs` denotes

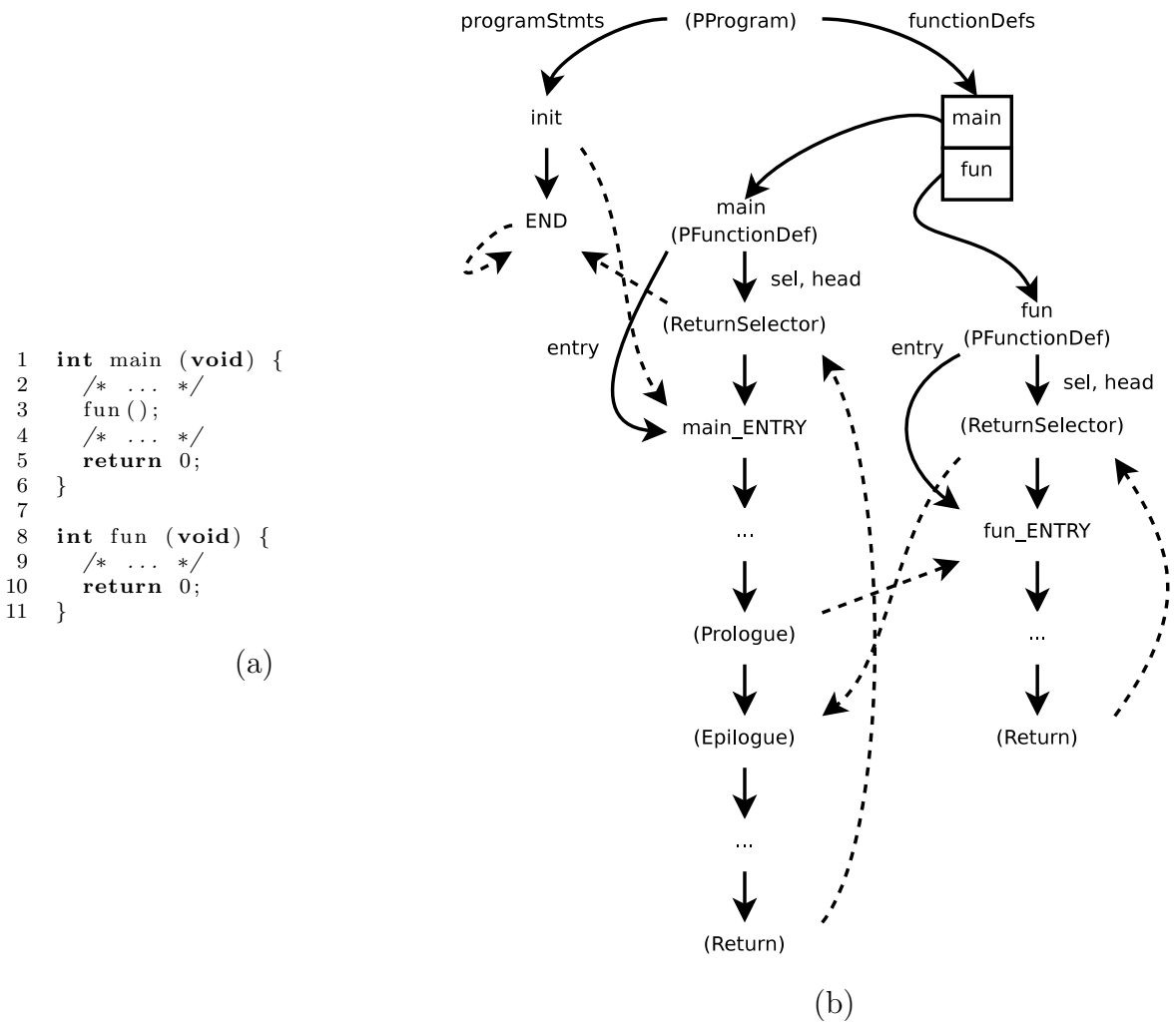



Figure 3.5: A simple program (a) and its `PProgram` structure (b).

the `functionDefs` attribute of the `PProgram` class. The various classes comprising a `PProgram` are summarized below.

Unlike the formal definition of a predicate program in Section 3.2, the implementation does not contain an explicit control-flow graph. Instead, it is composed of a collection of statements (`PStmt`s) with the following attributes:

- `next`: a pointer to the syntactic successor
- `dests`: a list of control-flow successors; written `dest` if there is only one successor
- `source`: a pointer to the corresponding statement in the CProgram

- `label`: the label of the corresponding statement in the CProgram, if any
- `parent`: a pointer to the enclosing function

Control-flow graph edges are denoted by the `dests` attribute. The syntactic ordering of statements is denoted by the `next` attribute. The types of `PStmt`s are summarized below.

- `AsmtPStmt`: a single assignment $p := ch(a, b)$.
- `PrllAsmtPStmt`: a parallel assignment as a list of `AsmtPStmt`s.
- `SkipPStmt`: a *skip* with `dest` = `next`.
- `GotoPStmt`: a *skip* with an arbitrary `dest`.
- `ConditionalBranchPStmt`: a two-way branch guarded by `cond` with `GotoPStmt`s `trueGoto` and `falseGoto` for the true and false branches, respectively.
- `SwitchBranchPStmt`: an $n$-way switch branch with a list of $n$ conditions $\texttt{cond}_i$ with a `GotoPStmt` $\texttt{goto}_i$ for each branch; there is one distinguished *default* condition $\texttt{cond}_{default}$ and goto $\texttt{goto}_{default}$.

As in a C program, statements in a `PProgram` are grouped into functions. The `PFunctionDef` class represents a function and has the following attributes:

- `head`: A pointer to the first statement in the body of the function. The body of a function is a graph of `PStmt`s whose edges are denoted by the `next` and `dest` attributes.
- `entry`: A pointer to the entry point of the function.
- `sel`: A pointer to the return selector: a multi-way branch pointing at each call site of the function.

- `returnIndexVar`: A variable denoting the index of a call site. All call sites for a given function are assigned a unique index.

- `returnValueVar`: A variable denoting the value returned by a non-void function.

The following is a summary of the classes representing function call and return:

- `FunctionCallPStmt`: A transition from a function call site to the function entry point. Composed of a `FunctionCallPrologue` and a `FunctionCallEpilogue`. Includes two attributes:
  - `callIndex`: the call site index.
  - `returnVar`: for a function call on the right-hand-side of an assignment, e.g., `y = f(x)`, `returnVar` is a variable storing the return value of the call (`y` in the example)

- `FunctionCallPrologue`: A parallel assignment representing the assignment of actual parameters to formal parameters and of `callIndex` to the callee's `returnIndexVar`

- `FunctionCallEpilogue`: A parallel assignment representing the assignment of the callee's `returnValueVar` to `returnVar`

- `ReturnPStmt`: A parallel assignment representing the assignment of a returned value, if any, to the enclosing function's `returnValueVar`. Transitions to the return selector of the function.

- `ReturnSelectorPStmt`: A multi-way branch targeting the `FunctionCallEpilogue` for each call site, guarded by `returnIndexVar`.

Finally, the `PProgram` class represents the entire predicate program. It provides three views of the program:

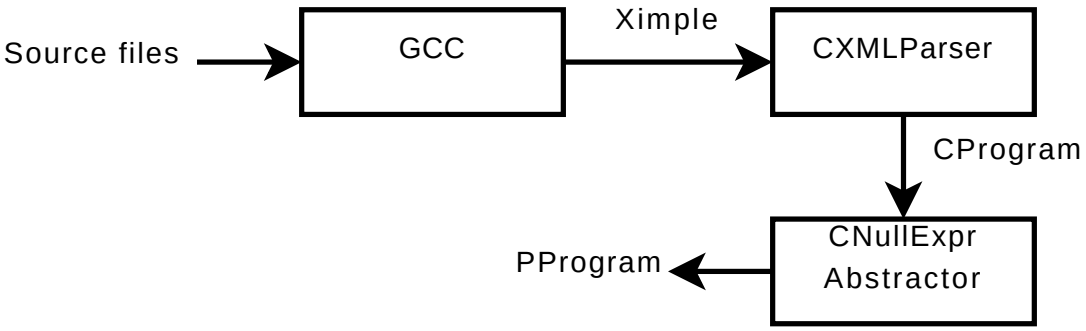



Figure 3.6: Ximple front-end architecture.

- `functionDefs`: A map from function name to `PFunctionDef`. Used, for example, to resolve the target of a function call and to find the `main` function.

- `statementList`: A list of all statements in the program. Used, for example, to iterate over each statement and compute a predicate abstraction.

- `labelledStatementMap`: A map from labels to the corresponding statements in the program. Used, for example, to find the target of a goto, or to resolve a label specified in the property being checked.

### 3.5.2 Implementation

Figure 3.6 shows the design of the Ximple front-end. First, a modified GCC compiler takes source and header files and produces a Ximple document. The next two components in the chain are summarized below.

**CXMLParser.** The Ximple document is passed to a SAX parser [6] which produces a corresponding `CProgram` object. As it parses the Ximple elements, it populates three tables corresponding to the three parts of a `CProgram`: a CFG table, a type table, and a declaration table. It constructs an AST for each Ximple subtree.

**CNullExprAbstractor.** The `CProgram` is transformed into a corresponding `PProgram` object. This class iterates over each function in the CFG table of the `CProgram` and constructs a corresponding `PFunctionDef` from the AST of the function.

### 3.5.3 Discussion

The Ximple front-end, not including the GCC modifications, took a single student about three weeks to implement. The majority of the implementation effort was spent in mapping the `CProgram` structure to the `PProgram` structure, which has several anachronisms due to it originally being designed around shortcomings of the ANTLR-based parser. The main advantage of using the Ximple representation is that it reduces the arbitrarily complex syntax of the C language to a relatively small set of XML tags with a simple structure. Using GCC as the first component of the front-end also takes care of preprocessing and (source-code) linking, which is a first step towards enabling analysis of large-scale programs in YASM.

## 3.6 Predicate Abstraction of Pointer Expressions

This section describes the design and implementation of several components which enable predicate abstraction of C expressions containing pointers. It begins with a background on the semantics of pointer expressions as defined by the C language and OldYASM followed by a generalized definition of WP which correctly handles pointers. It then defines the "logical memory model" which improves upon OldYASM's semantics of pointer expressions. This is followed by an overview of the implementation of the new components. The section concludes with an illustration of the revised predicate abstraction process and a discussion.

### 3.6.1 Background

The presentation of pointer expression semantics below roughly follows that of Chaki et al [18].

**C semantics.** In C, memory is organized as a finite array of words—a word is typically four bytes. Each word has an *address* which is its index in the array. For each

variable declaration, C allocates one or more contiguous words and associates the variable with the index of the first word. Assume the only datatype is integer. Let $V$ be the set of program variables, $M : \mathbb{N} \to \mathbb{Z}$ map an address to an integer, and $B : V \to \mathbb{N}$ map a variable to an address. C distinguishes between *lvals* and *rvals*: lvals are expressions which can appear on the left hand side of an assignment statement, and rvals are those which can appear on the right. The grammar for C expressions is below, where lvals are defined by $LV$ and rvals by $RV$:

$$LV := V \mid *RV$$
$$RV := \mathbb{Z} \mid LV \mid \&LV \mid RV \ op \ RV$$

The semantics of rvals are defined by the evaluation function *eval*:

$$eval(rv) = \begin{cases} rv & \text{if } rv \in \mathbb{Z} \\ M(B(rv)) & \text{if } rv \in V \\ M(eval(rv')) & \text{if } rv = *rv' \\ M(rv'') & \text{if } rv = \&rv' \text{ and } eval(rv') = M(rv'') \\ eval(rv_1) \ op \ eval(rv_2) & \text{if } rv = rv_1 \ op \ rv_2 \end{cases}$$

The semantics of an assignment $lv := rv$ are defined in terms of the pre and post state of the memory, $M$ and $M'$ respectively. If $lv \in V$, then

$$M'(n) = \begin{cases} eval(rv) & \text{if } n = B(lv) \\ M(n) & \text{otherwise} \end{cases}$$

and if $lv = *rv'$ for some rval $rv'$, then

$$M'(n) = \begin{cases} eval(rv) & \text{if } n = eval(rv') \\ M(n) & \text{otherwise.} \end{cases}$$

*Arrays.* The meaning of an array expression, e.g., $a[5]$, is obtained by first translating to the equivalent pointer expression, e.g., $*(a+5)$. An array variable, e.g., $a$ which is declared `int a[6]`, has a special meaning as an rval:

$$eval(rv) = B(rv) \text{ if } rv \in V \text{ and } rv \text{ is an array variable.}$$

It is illegal for an array variable to appear without a subscript on the left hand side of an assignment.

**OldYASM semantics.** OldYASM assumes all program variables are integers and lacks the notion of an address. Memory is modeled as a map $M : V \to \mathbb{Z}$. The supported syntax of C expressions is as follows:

$$LV := V$$
$$RV := \mathbb{Z} \mid LV \mid RV \ op \ RV$$

The semantics of rvals are

$$eval(rv) = \begin{cases} rv & \text{if } rv \in \mathbb{Z} \\ M(rv) & \text{if } rv \in V \\ eval(rv_1) \ op \ eval(rv_2) & \text{if } rv = rv_1 \ op \ rv_2 \end{cases}$$

and the semantics of an assignment $lv := rv$ are

$$M'(v) = \begin{cases} eval(rv) & \text{if } v = lv \\ M(v) & \text{otherwise.} \end{cases}$$

Notice the lack of support for pointer operators. As a workaround, OldYASM rewrites the C expression `&x` (address of `x`) as a variable `addr_x` and `*x` (dereference `x`) as a variable `ptr_x`, which yields unsound results. For example, the model interprets the assignment `*x = 5` as $ptr\_x = 5$, i.e., an update to the fictitious variable `ptr_x`. Consider the program

```
1  int i, *ip;
2  i = 5;
3  ip = &i;
4  if (*ip != 5)
5    ERROR: ;
```

(a)

```
1  int i, ptr_ip, addr_i;
2  i = 5;
3  ip = addr_i;
4  if (ptr_ip != 5)
5    ERROR: ;
```

(b)

Figure 3.7: A simple pointer program (a) and its representation in OldYASM (b).

in Figure 3.7(a) in which the line labeled `ERROR` is unreachable. OldYASM rewrites the program as shown in Figure 3.7(b), in which `ERROR` may be reachable since `ptr_ip` may be non-deterministically initialized to 5.

**General WP.** In the presence of pointers, the simple definition of WP is incorrect. To see why, consider the formula

$$wp(*p := 1, (i > 0)) = (i > 0)[1/*p] = (i > 0)$$

with declarations `int i, *p`. Intuitively, this says that the assignment to $*p$ cannot affect $i$. This definition of WP fails to capture the case where `*p` and `i` are aliases (`p == &i`), so an assignment to `*p` does in fact modify `i`.

To rectify this, we use the generalized WP defined in [13], adapted from Morris' general axiom of assignment [40]. First, define *mutable* to be one of the following expressions:

- A non-array variable, e.g., `i` or `p` with `int i, *p`.
- An element of an array, e.g., `a[1]` with `int a[]`.
- A dereference of a pointer, e.g., `*p` with `int *p`.

A mutable roughly corresponds to an *lval* in C: an expression which can appear on the LHS of an assignment. Next, define the *conditional substitution* for a predicate $p$, assignment $x := e$, and mutable $y$ in $p$ as

$$p[x, e, y] = (\&x = \&y \wedge p[e/y]) \vee (\&x \neq \&y \wedge p).$$

The first disjunct captures the case where $x$ and $y$ are aliased, so $y$ obtains the value $e$ after the assignment. The second disjunct captures the case where $x$ and $y$ are not aliased, so the assignment to $x$ leaves $y$ unchanged. Now we define the general WP, where $y_1, \ldots, y_n$ are the mutables appearing in $p$, as

$$wp(x := e, p) = p[x, e, y_1][x, e, y_2] \cdots [x, e, y_3].$$

For the original example, we have

$$\begin{aligned} & wp(*p := 1, i > 0) \\ = \; & (i > 0)[*p, 1, i] \\ = \; & (\& * p = \& i \wedge (i > 0)[1/i]) \vee (\& * p \neq \& i \wedge (i > 0)) \\ = \; & (p = \& i \wedge \mathbf{true}) \vee (p \neq \& i \wedge (i > 0)). \end{aligned}$$

Intuitively, this says that if $p$ points to $i$, the assignment makes $(i > 0)$ true; otherwise, the value of $(i > 0)$ depends on its value before the assignment.

### 3.6.2 Logical Memory Model

This model is derived from that of Chaki et al. [18] with additional input from Arie Gurfinkel and Tom Hart. It organizes memory as a set of disjoint arrays, called *objects*, and associates each program variable with a unique object. This model supports the full syntax of lvals and rvals. Basic definitions are as follows:

- An *object* is an array of cells.
- An *address* is a tuple (*object*, *offset*), where *offset* is a natural number.
- A *value* is an integer or an address.

The logical memory model (LMM) consists of the following parts, where $V$ is the set of program variables:

- A set of objects $Obj = \{obj_v | v \in V\}$.
- A set of addresses $Addr = Obj \times \mathbb{Z}$.
- A set of values $Val = Addr \cup \mathbb{Z}$.
- A referencing function $ref : Addr \to Val$.
- An object map $obj : V \to Obj$. The map is subject to the constraint $\forall u, v \in V : obj(u) = obj(v) \Rightarrow u = v$. That is, every variable is associated with a unique object.

Intuitively, this model represents a variable as a unique name of an object, as indicated by the map $obj$. Define an additional function $base(v : V) = (obj(v), 0)$, which returns the first, or "base", address of the object pointed to by the given variable. The semantics of rvals are as follows:

$$eval(rv) = \begin{cases} rv & \text{if } rv \in \mathbb{Z} \\ ref(base(rv)) & \text{if } rv \in V \text{ and } rv \text{ is of type } \texttt{int} \text{ or } \texttt{int*} \\ base(rv) & \text{if } rv \in V \text{ and } rv \text{ is of type } \texttt{int[]} \\ rv'' & \text{if } rv = \&rv' \text{ and } eval(rv') = ref(rv'') \\ ref(eval(rv')) & \text{if } rv = *rv' \\ eval(rv_1)\ op\ eval(rv_2) & \text{if } rv = rv_1\ op\ rv_2 \text{ if } eval(rv_1), eval(rv_2) \in Z \end{cases}$$

The type of $eval(rv)$ for $rv \in V$ depends on the type of $rv$: if $rv$ is of type `int*` or `int[]`, $eval(rv)$ is of type $Addr$; if $rv$ is of type `int`, $eval(rv)$ is of type $\mathbb{Z}$.

The semantics of arithmetic and comparison operations on addresses are as follows:

- $eval((obj, off) + rv) = (obj, off + eval(rv))$ and similarly for subtraction. Address arithmetic only changes the offset of an address; there is no operation to change the object of an address.

- $eval((obj_1, off_1) < (obj_2, off_2)) = (obj_1 = obj_2 \wedge off_1 < off_2)$ and similarly for equality. Notice that the comparison holds iff the two addresses refer to the same object.

Finally, the semantics of an assignment $lv := rv$ are defined in terms of the pre and post state of the referencing function $ref'$. If $lv \in V$ then

$$ref'((o, f)) = \begin{cases} eval(rv) & \text{if } o = obj(lv) \\ ref((o, f)) & \text{otherwise} \end{cases}$$

and if $lv = *rv'$ for some rval $rv'$ then

$$ref'((o, f)) = \begin{cases} eval(rv) & \text{if } (o, f) = eval(rv') \\ ref((o, f)) & \text{otherwise} \end{cases}$$

As before, the meaning of an array expression is obtained by first translating to the equivalent pointer expression.

### 3.6.3 Implementation

This subsection describes the implementation of the components which together enable predicate abstraction of pointer expressions. Figure 3.8 shows the architecture of the new components. The system takes as input a statement $s$ and a predicate $p$, both of which may contain pointer expressions, and first computes the generalized WP, $\phi = wp(s, p)$, using `MorrisWPComputer`. Then, YASM constructs a strengthening of $\phi$ by issuing queries and stack operations to check the validity of $c \Rightarrow \phi$ where $c$ is a cube. The classes `CTheoremProver` and `CToAddrModel` translate between C expressions and their equivalent expressions in CVCL's input language according to the logical memory model and symbol table. Each class is described in more detail below.

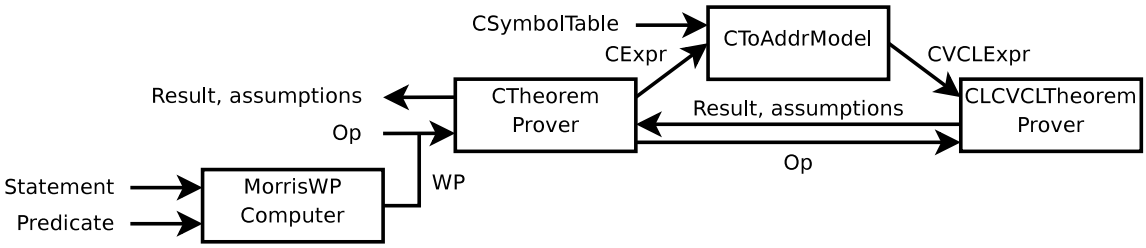



Figure 3.8: Extended predicate abstraction architecture.

`MorrisWPComputer`. Computes the general WP for a given statement and predicate. Note that the general WP contains at most $2^n$ disjuncts for a predicate with $n$ mutables, one for each possible combination of aliases between the LHS of the assignment and the mutables. The implementation includes an optimization which prunes a disjunct ($\&x = \&y \wedge \cdot$) if the types of $x$ and $y$ are different, in which case they cannot possibly be aliases.

`CTheoremProver`. Implements the `ITheoremProver` interface. Stack and query operations are passed to `CLCVCLTheoremProver` in a straightforward manner; `CToAddrModel` is used to rewrite C expressions into CVCL expressions conforming to the LMM before passing them to the theorem-prover. Returns the result of the previous query and, if the query was valid, the list of assumptions used in its proof.

`CToAddrModel`. Rewrites a C expression according to the LMM and the given symbol table. Objects are represented as integers; the object map, $obj$, is implemented as a map which associates each variable in the symbol table with a unique positive integer.

The PIL (CVCL's presentation input language) syntax we use in rewriting C expressions is as follows:

- Declaration: a symbol `a` is declared as type `t` as `a:t`. The built-in types which we used are below.
- Integer: `INT`.
- Array: `ARRAY ti OF tv` where `ti` is the index type and `tv` is the element type.

- Function: `td -> tr`, where `td` and `tr` are the domain and range types, respectively.
- Tuple: `[t1,t2,...]`, where `t`*i* is a type. The first element of a tuple `u` is expressed as `u.0`, the second as `u.1`, and so on.

To implement the LMM on top of CVCL's built-in theories of integers and arrays, CVCL is initialized as follows:

1. Declare an address type: `ADDR:[INT,INT]`.
2. Declare *scalar-ref* functions mapping the *object* and *offset* parts of an `ADDR` to a value. For example, `INT_REF : ADDR -> INT` maps an `ADDR` to an integer value.
3. Declare *array-ref* functions mapping the *object* part of an `ADDR` to an `ARRAY`. For example, `INT_ARRAY_REF : ADDR -> ARRAY INT OF INT` maps an `ADDR` to an array with integer indices and integer elements.

Using these CVCL declarations, `CToAddrModel` recursively rewrites a C expression to a CVCL expression. Rewriting of the base cases is summarized below. Assume the following declarations: `int i, *p, *q, a[]` and the following address map: $(\texttt{i}, 1), (\texttt{p}, 2), (\texttt{a}, 3), (\texttt{q}, 4)$. The base cases are rewritten as follows:

- `&i`: `[1,0]`.
- `i`: `INT_REF([1,0])`.
- `p`: `ADDR_REF([2,0])`.
- `*p`: `INT_REF(ADDR_REF([2,0]))`.
- `a`: `INT_ARRAY_REF([3,0])`.
- `a[3]`: `(INT_ARRAY_REF([3,0]))[3]`.
- `&a[3]`: `[[3,0].0, ([3,0].1) + 3]`, which reduces to `[3,3]`.

```
1  void main (void) {
2    int *x, y;
3    y = 0;
4    x = &y;
5    if (*x > 0)
6      { ERROR: ; }
7  }
```

Figure 3.9: Simple pointer program.

Whether the *scalar-* or *array-ref* function is used is determined by the type of the argument, which is provided by the symbol table. An exception is thrown if a given expression has a type violation; e.g., `*i` yields an exception since `i` is not a pointer. Pointer arithmetic and comparison are rewritten following the LMM:

- `p + 3`: `[ADDR_REF([2,0]).0, (ADDR_REF([2,0]).1) + 3]`. Similarly for subtraction.

- `p < q`: `((ADDR_REF([2,0]).0) = (ADDR_REF([4,0]).0)) AND ((ADDR_REF([2,0]).1) < (ADDR_REF([4,0]).1))`. Similarly for equality.

All other operators, e.g., integer arithmetic and comparison, and operands, e.g., integer literals, are left as-is.

### 3.6.4 Illustration

Now we show how the extension translates a query over C expressions to an equivalent CVCL query, according to the LMM. Consider the program shown in Figure 3.9 and suppose we are checking for the reachability of the line labeled `ERROR`. After one CEGAR iteration, predicate generation produces the predicate $*x > 0$ to track the branch condition of the `if` statement. To obtain the next predicate abstraction requires first computing the WP of this new predicate over each of the program statements. The interesting case is $wp(x = \&y, *x > 0)$. The mutables appearing in $*x > 0$ are $x$ and $*x$. The computation of the generalized WP is shown in Figure 3.10.

$$
\begin{aligned}
& wp(x = \&y, *x > 0) \\
&= (*x > 0)[x, \&y, x][x, \&y, *x] \\
&= ((\&x = \&x \wedge (*x > 0)[\&y/x]) \vee (\&x \neq \&x \wedge (*x > 0)))[x, \&y, *x] && \text{Identity } \&x = \&x \\
&= ((*x > 0)[\&y/x])[x, \&y, *x] \\
&= (*\&y > 0)[x, \&y, *x] && \text{Identity *\&y = y} \\
&= (y > 0)[x, \&y, *x] \\
&= (\&x = \& * x \wedge (y > 0)[\&y, *x]) \vee (\&x \neq \& * x \wedge (y > 0)) && x \text{ is of type } \texttt{*int} \\
&= \&x \neq \& * x \wedge (y > 0) \\
&= y > 0
\end{aligned}
$$

Figure 3.10: Example generalized WP computation.

Notice the use of type information provided by the symbol table to determine that $\&x \neq \& * x$: $x$ is of type `*int` whereas $*x$ is of type `int`, so the addresses of these two expressions cannot be equal under the LMM. Our current predicate set is simply $\{*x > 0\}$ and, since $y > 0$ is not in this set, YASM now attempts to compute a strengthening of $y > 0$. Suppose the address map generated by `CToAddrModel` is $\{(x, 1), (y, 2)\}$ and that the first query checked by `CTheoremProver` is $*x > 0 \Rightarrow y > 0$. `CToAddrModel` translates the query as follows: `INT_REF(ADDR_REF([1,0])) > 0 ⇒ INT_REF([2,0]) > 0`. `CLCVCLTheoremProver` returns *invalid*, intuitively because there are no constraints between `[1,0]` and `[2,0]`, and similarly for $\neg(*x > 0) \Rightarrow y > 0$. Proceeding in this fashion, statement `x = &y` is abstracted as $(*x > 0) := *$, i.e., in this abstraction, the value of $(*x > 0)$ is unknown after the statement `x = &y`. In a subsequent CEGAR iteration, YASM adds predicate $y > 0$ to the set by recomputing the WP as shown above. YASM them abstracts `x = &y` using the new predicate set $\{*x > 0, y > 0\}$, obtaining the abstraction $(*x > 0) := \mathit{choice}(y > 0, \neg(y > 0)), (y > 0) := \mathit{choice}(y > 0, \neg(y > 0))$, which is precise enough to determine the (un)reachability of `ERROR`.

### 3.6.5 Discussion

**Differences from the C memory model.** An important distinction between the C memory model and the LMM is that C treats memory as a single array of cells. So, whereas objects are disjoint in the LMM, objects are laid out contiguously in C. Therefore, it is possible in C to access cells of one object by accessing another object with a sufficiently large offset. However, since pointer arithmetic in the LMM cannot change the object to which a pointer refers, YASM may incorrectly analyze programs which use memory in this manner.

C also permits other unstructured uses of memory. In particular, C permits *type casting* in which memory declared as one type can be evaluated as a different type. A common example of this is casting an array of 4 `char`s to an `int`. This often occurs when data is copied "off the wire", e.g., from a network socket, byte-by-byte into an array and then subsequently used as more specific datatypes. Our implementation of the LMM currently does not support type casting, specifically because CVCL itself does not allow for an array of bytes to be treated as a (mathematical) integer. It may be possible to overcome this limitation if we model integers as 32-bit words instead of mathematical integers.

**Limitations and future work.** Our implementation of `MorrisWPComputer` uses a basic optimization to prune disjuncts where two mutables do not have the same type. It may be beneficial to use alias analysis to further prune disjuncts where two mutables are known to be unaliased at a particular program statement. In general, any external information, e.g., as provided by a dataflow analysis, could be useful in reducing the WP before computing a strengthening. For example, [34] uses octagon invariants, specifying arithmetic constraints over program variables, to reduce WP expressions.

The `CTheoremProver` implementation currently lacks support for language features such as structs, unions, and bitwise operations. CVCL provides support for these constructs, so it is possible to extend YASM to support them as well. The first steps are to

extend the `Expr` language to cover these constructs and to specify axioms and rewriting rules for operators such as field selection, in the case of structs and unions.

## 3.7 Related Work

**Memory models in SMCs**

*Slam and Blast.* Slam [14] and Blast [16] are two CEGAR SMCs for C which use a memory model similar to the LMM with an important exception that is not documented. Specifically, whereas YASM models an address as a pair $(object, offset)$, these SMCs identify an address with an object. Pointer arithmetic, which in YASM modifies the offset component of an address, has no effect in the Slam/Blast model. Likewise, array indexes are discarded, so all array offsets, e.g., `a[1]`, `a[2]`, and `a[3]`, are interpreted as the first element of the array, i.e., `a[0]`. Since all information about pointer offsets is discarded, these SMCs are unsuitable for checking buffer overflows.

*Copper.* Copper [18] is a CEGAR SMC for C which defines a memory model similar to the LMM used in YASM. It defines two additional functions specifically for buffer overflow detection. Function $alloc : Obj \rightarrow \mathbb{Z}$ maps an object to its allocated size, i.e., the number of cells in the underlying array. Function $size : Obj \rightarrow \mathbb{Z}$ maps an object to the number of cells occupied from the beginning of the array up to and including the first occurence of the "null" character, i.e., *size* returns the length of the null-terminated string occupying a given object.

These functions are used in Copper's transformation rules to check for buffer overflows in the string functions of the standard C library. For example, the function call `strcpy(p,q)` is translated as

$$assume(size(obj(p)) = size(obj(q)) - (q - base(obj(q))) + (p - base(obj(p)))).$$

Intuitively, this models the behaviour of `strcpy` as copying the substring pointed to by

`q` into the array offset pointed by `p`. Additional rules are defined for the string functions `strncpy`, `strcat`, and `strncat`.

Copper also defines transformation rules for the dynamic memory allocation functions, `malloc` and `free`. For example, the statement

$$\texttt{p} = \texttt{malloc(15)}$$

is translated as

$$assume(alloc(obj(p))) = 15.$$

This is used in conjunction with instrumentation to check the safety of string operations. For example, an assertion is inserted after the `strcpy` call above:

$$assert(size(obj(p))) \leq alloc(obj(p))$$

which asserts that the (string-)length of `p` does not exceed its allocated size.

The advantage of using high-level semantics of standard library functions is that it enables Copper to avoid the potentially expensive analysis of the bodies of these functions. However, notice that the interpretation of `strcpy` shown above ignores the *value* of the string copied from `q` to `p`. If a property depends on these values, Copper may produce unsound results. This may explain Copper's poor performance on the Zitser suite [53], as documented in [18], since many of the testcases depend on the values of specific string elements.

**Intermediate representations for C**

*CIL.* CIL (C Intermediate Language) [41] produces a canonical representation of a C program using a small subset of the C language. For example, all loop statements are transformed into `while` loops; statements with multiple side-effects, e.g., `x = y++`, are transformed into single side-effect statements, e.g., `x = y; y = y + 1`; declaration and

```
1  void main(void)
2  {
3    char buf[2];
4    char *buf_ext;
5    char *dest;
6    char input[10];
7    char *src;
8
9    buf_ext = &(buf[1]);
10   dest = &(buf[0]);
11   src = &(input[0]);
12   while (1) {
13     if (dest > buf_ext)
14       ERROR:;
15     *dest = *src;
16
17     if (*src == 0)
18       goto END;
19
20     dest = dest + 1;
21     src = src + 1;
22   }
23   END: ;
24 }
```

Figure 3.11: A string copy loop.

initialization statements are separated, e.g. `int i = 5;` becomes `int i; i = 5;`. CIL also provides AST and CFG views of a given program. As such, CIL simplifies subsequent program analysis because it reduces the number of forms to be handled. Unfortunately, we were unable to make full use of CIL since its programmatic interface is only available in Ocaml.

*GCC-XML.* GCC-XML [3], developed by Brad King, is an extension for the GCC compiler which produces an XML "description" of a C++ program. It differs from the Ximple representation in that GCC-XML discards function bodies and only produces information about declarations, i.e., variable and function declarations. The GCC-XML code may be a useful guide for extending Ximple to handle C++ constructs such as classes and templates.

## 3.8 Conclusion

**Current Status.** YASM can successfully check the reachability of the `ERROR` statement in the program shown in Figure 3.11. However, at larger buffer sizes the strengthening

computation (apparently) does not terminate, due to a bug in the theorem-prover component. It is unclear whether the bug is in CVCL or in our interface. In the future we plan to run YASM on the Verisec benchmark in order to uncover bugs in our implementation and to identify opportunities for optimization.

**Future Work.** Beyond debugging YASM and evaluating it on our benchmark suite, there are many possibilities for future development. As mentioned above, using the Ximple interface enables YASM to analyze large-scale programs. Since large programs tend to be organized into relatively small functions, the Ximple representation could enable compositional analysis by not only storing the syntactic information about the source program, but the results of previous analyses as well. For example, Ximple could be used to store pre- and post-condition annotations for each function. From our initial work with YASM, we found that the strengthening computation dominates the overall cost of analysis. As such, it is likely beneficial to develop strategies for either reducing the number or complexity of queries sent to the theorem prover. For example, SLAM [15] uses a very simple strategy which appears to be effective for real-world programs: limit the size of cubes to contain at most three conjuncts. It may also be beneficial to modify the theorem-prover or the interface to exploit patterns that occur in the queries generated by YASM. Typically the queries are of trivial complexity, e.g., $x > 0 \wedge y > 0 \Rightarrow x + y > 0$, and may be amenable to much simpler techniques, such as pattern matching, than full-blown theorem-proving.

# Chapter 4

# Conclusion

## 4.1 Summary

We conducted an assessment of several publically available CEGAR SMC implementations and found that all the tools had problems parsing and interpreting real-world code and generally performed poorly in analyzing the testcases. Based on our findings, we constructed a benchmark containing 298 testcases derived from 22 vulnerabilities in 12 real-world programs. We evaluated the benchmark on the SMC SatAbs and found that it produced a wide range of behaviour from the tool. The evaluation confirmed that our testcase construction process produces a set of testcases of increasing complexity. The evaluation also showed that SatAbs' performance is independent of buffer size in analyzing certain forms of safe testcases. We defined a complexity measure, trace length, based on the program dependency graph and implemented a CodeSurfer plugin to compute the measure for a given program. Our evaluation of the measure shows a correlation between trace length and CEGAR analysis complexity.

We implemented several extensions to YASM to support the analysis of real-world C programs containing pointers and arrays. To the front-end we added a parser for an XML representation of C programs. The back-end was extended with an interface

for a command-line theorem prover. Finally, the predicate abstraction component was extended to handle pointer expressions. The extensions enable YASM to check for buffer overflows in a program containing a string-copy loop.

## 4.2 Limitations

Given the results of our benchmark evaluation, it is unclear whether CEGAR is a suitable analysis technique for buffer overflow analysis. However, it is one of the few known techniques which is capable of *verification*, which is one of our analysis requirements. In certain cases CEGAR can efficiently verify that a program is safe.

On the other hand, our insistence on this requirement may be unfounded: it is possible that an analysis that is only capable of efficient overflow *detection* is sufficient to ensure security. Our focus on SMC was biased by our work on YASM and we might have otherwise selected a different technique to study, such as directed testing or explicit-state model-checking, which have been shown to be effective at buffer overflow detection (but not verification) [39, 17].

## 4.3 Future Plans

Our initial assessment of existing SMCs and development of the benchmark was done in isolation, that is, without consulting the developers of the tools we were using. We are currently working with the authors of SatAbs to debug the testcases (and their tool). In the future, we plan to discuss the results of our assessment with the rest of the tool developers in the hopes of expanding the applicability of the benchmark.

The majority of the work in creating the benchmark was spent on manual simplification of the testcases. It may be possible to implement an automatic testcase construction procedure using a technique for systematically modifying program syntax, similar to that of Kratkiewicz [35]. This would provide a clearer, less labour-intensive, and more easily

replicable construction process.

It is clear from the results of the benchmark and our work on YASM that buffer-size-dependence is the most significant limitation to CEGAR analysis of buffer overflows. A promising approach to overcoming this limitation is efficient CEGAR-based analysis of loops, such as [36, 37]. The existing techniques for loop analysis are only able to handle toy programs, so this is an open area for research.

# Bibliography


[1] "ANTLR Parser Generator". http://www.antlr.org/.

[2] "Counting Source Lines of Code (SLOC)". http://www.dwheeler.com/sloc/.

[3] "GCC-XML". http://www.gccxml.org/.

[4] "GNU, the GNU Compiler Collection". http://gcc.gnu.org/.

[5] "Java Native Interface". http://java.sun.com/j2se/1.4.2/docs/guide/jni/.

[6] "SAX: Simple API for XML". http://saxproject.org/.

[7] "SWIG: Simplified Wrapper and Interface Generator". http://www.swig.org/.

[8] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools, Chapter 10*. Addison Wesley, 1988.

[9] N. Amla, R. P. Kurshan, K. L. McMillan, and R. Medel. "Experimental Analysis of Different Techniques for Bounded Model Checking". In *Proceedings of 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, pages 34–48, 2003.

[10] D. Atiya, N. Catao, and G. Lüettgen. "Towards a Benchmark for Model Checkers of Asynchronous Concurrent Systems". In *Fifth International Workshop on Automated Verification of Critical Systems: AVOCs*, 2005.

[11] G. Balakrishnan, R. Gruian, T. W. Reps, and T. Teitelbaum. "CodeSurfer/x86-A Platform for Analyzing x86 Executables". In *Proceedings of 14th International Conference on Compiler Construction (CC'05)*, pages 250–254, 2005.

[12] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. "Thorough Static Analysis of Device Drivers". In *Proceedings of EuroSys'06*, pages 73–85. ACM Press, 2006.

[13] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. "Automatic Predicate Abstraction of C Programs". In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming language design and implementation (PLDI '01)*, pages 203–213, New York, NY, USA, 2001. ACM Press.

[14] T. Ball, A. Podelski, and S. Rajamani. "Boolean and Cartesian Abstraction for Model Checking C Programs". In *Proceedings of 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *LNCS*, pages 268–283, April 2001.

[15] T. Ball and S. Rajamani. "The SLAM Toolkit". In *Proceedings of 13th International Conference on Computer-Aided Verification (CAV'01)*, volume 2102 of *LNCS*, pages 260–264, July 2001.

[16] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. "Checking Memory Safety with Blast". In *Proceedings of Formal Aspects in Software Engineering (FASE'05)*, pages 2–18. Springer, 2005.

[17] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. "EXE: Automatically Generating Inputs of Death". In *Proceedings of 13th ACM Conference on Computer and Communications Security (CCS'06)*, pages 322–335. ACM Press, 2006.

[18] S. Chaki and S. Hissam. "Certifying the Absence of Buffer Overflows". Technical Report CMU/SEI-2006-TN-030, SEI, CMU, September 2006.

[19] M. Chechik, B. Devereux, and A. Gurfinkel. "$\chi$Chek: A Multi-Valued Model-Checker". In *Proceedings of 14th International Conference on Computer-Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 505–509, Copenhagen, Denmark, July 2002. Springer.

[20] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[21] E. Clarke and D. Kroening. "ANSI-C Bounded Model Checker". User manual, Carnegie Mellon University, August 2006.

[22] E. Clarke, D. Kroening, and F. Lerda. "A Tool for Checking ANSI-C Programs". In *Proceedings of 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *LNCS*, pages 168–176. Springer, March 2004.

[23] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. "Predicate Abstraction of ANSI–C Programs using SAT". *Formal Methods in System Design (FMSD)*, 25:105–127, September–November 2004.

[24] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. "SATABS: SAT-Based Predicate Abstraction for ANSI-C". In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *LNCS*, pages 570–574. Springer Verlag, 2005.

[25] CVE — Common Vulnerabilities and Exposures. http://cve.mitre.org/.

[26] M. B. Dwyer, S. Person, and S. G. Elbaum. "Controlling Factors in Evaluating Path-sensitive Error Detection Techniques". In *Proceedings of SIGSOFT Conference on Foundations of Software Engineering (FSE'06)*, pages 92–104, 2006.

[27] D. Evans and D. Larochelle. "Improving Security Using Extensible Lightweight Static Analysis". *IEEE Software*, 19(1):42–51, 2002.

[28] P. Godefroid. "Software Model Checking: The VeriSoft Approach". Technical Report ITD-03-44189G, Bell Labs, March 2003.

[29] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. "SYNERGY: A New Algorithm for Property Checking". In *Proceedings of SIGSOFT Conference on Foundations of Software Engineering (FSE'06)*, pages 117–127. ACM Press, 2006.

[30] A. Gurfinkel and M. Chechik. "Why Waste a Perfectly Good Abstraction?". In *Proceedings of 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, volume 212-226 of *LNCS*, page 3920, Vienna, Austria, April 2006. Springer.

[31] A. Gurfinkel, O. Wei, and M. Chechik. "YASM: A Software Model-Checker for Verification and Refutation". In *Proceedings of 18th International Conference on Computer-Aided Verification (CAV'06)*, volume 170-174 of *LNCS*, page 4144, Seattle, WA, August 2006. Springer.

[32] K. Havelund and T. Pressburger. "Model Checking Java Programs Using Java Pathfinder". *International Journal on Software Tools for Technology Transfer*, 1999.

[33] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. "Lazy Abstraction". In *Proceedings of 29th Symposium on Principles of Programming Languages (POPL'02)*, pages 58–70. ACM, January 2002.

[34] H. Jain, F. Ivancic, A. Gupta, I. Shlyakhter, and C. Wang. "Using Statically Computed Invariants Inside the Predicate Abstraction and Refinement Loop". In *Proceedings of 18th International Conference on Computer-Aided Verification (CAV'06)*, pages 137–151. Springer, 2006.

[35] K. Kratkiewicz and R. Lippmann. "Using a Diagnostic Corpus of C Programs to Evaluate Buffer Overflow Detection by Static Analysis Tools". In *Proceedings of 2005 Workshop on the Evaluation of Software Defect Detection Tools (BUGS'05)*, June 2005. http://www.cs.umd.edu/∼pugh/BugWorkshop05/.

[36] D. Kroening, A. Groce, and E. Clarke. "Counterexample Guided Abstraction Refinement via Program Execution". In *Proceedings of Int. Conf. on Formal Eng. Methods*, pages 224–238, November 2004.

[37] D. Kroening and G. Weissenbacher. "Counterexamples with Loops for Predicate Abstraction". In *Proceedings of 18th International Conference on Computer-Aided Verification (CAV'06)*, volume 4144 of *LNCS*, pages 152–165. Springer Verlag, 2006.

[38] R. Lemos. "Browsers Feel the Fuzz". www.securityfocus.com/news/11387, April 2006.

[39] R. Majumdar and R.-G. Xu. "Directed Test Generation Using Symbolic Grammars". In *Proceedings of Joint 15th European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/SIGSOFT FSE'07)*, pages 553–556, 2007.

[40] J. M. Morris. "A General Axiom of Assignment". *Theoretical Foundations of Programming Methodology*, pages 25–34, 1982.

[41] G. Necula, S. McPeak, S. Rahul, and W. Weimer. "CIL: Intermediate Language and Tools for Analsysis and Transformation of C Programs". In *Proceedings of 11th International Conference on Compiler Construction (CC'02)*, volume 2304 of *LNCS*, pages 213–228, Grenoble, France, April 2002. Springer.

[42] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. "CCured: Type-Safe Retrofitting of Legacy Software". *ACM TOPLAS*, 27(3):477–526, 2005.

[43] A. One. "Smashing the Stack for Fun and Profit", 1995. http://insecure.org/stf/smashstack.html.

[44] R. Pelanek. "Model Classifications and Automated Verification". In *Formal Methods for Industrial Critical Systems (FMICS'07)*, pages 15–30, 2007.

[45] K. Sen, D. Marinov, and G. Agha. "CUTE: a Concolic Unit Testing Engine for C". In *Proceedings of the 13th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*, pages 263–272. ACM Press, 2005.

[46] S. E. Sim, R. C. Holt, and S. M. Easterbrook. "On Using a Benchmark to Evaluate C++ Extractors". In *Proceedings of 10th International Workshop on Program Comprehension (IWPC'02)*, pages 114–126. IEEE Computer Society, 2002.

[47] F. Tip. "A Survey of Program Slicing Techniques". *Journal of Programming Languages*, 3(3), 1995.

[48] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. "ITS4: A Static Vulnerability Scanner for C and C++ Code". In *Proceedings of 16th Annual Computer Security Applications Conference (ACSAC'00)*, pages 257–, 2000.

[49] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities". In *Proceedings of the Network and Distributed System Security Symposium (NDSS'00)*, pages 3–17, February 2000.

[50] J. Whaley and M. S. Lam. "Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams". In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming language design and implementation (PLDI'04)*, pages 131–144. ACM, 2004.

[51] J. Wilander and M. Kamkar. "A Comparison of Publicly Available Tools for Static Intrusion Prevention". In *Proceedings of 7th Nordic Workshop on Secure IT Systems*, pages 68–84, November 2002.

[52] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. "The SPLASH-2 Programs: Characterization and Methodological Considerations". In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*, pages 24–36, 1995.

[53] M. Zitser, R. Lippmann, and T. Leek. "Testing Static Analysis Tools Using Exploitable Buffer Overflows from Open Source Code". In *Proceedings of SIGSOFT Conference on Foundations of Software Engineering (FSE'04)*, pages 97–106. ACM Press, 2004.

# Appendix A

# Benchmark evaluation data

The results of the benchmark evaluation are presented in the table on the following pages. The results have been filtered, as discussed in Section 2.7, to only include claims which were checked successfully at all buffer sizes. Each row of the table shows the result for a single testcase. For example, the first row shows the results for vulnerability CVE-2004-0940 from Apache, referred to as "Apache 1", in the testcase `iter1_prefixLong_arr`. The next six columns show the measurements and performance results for the safe variant of the testcase, while the following six columns are for the unsafe variant. LOC shows the number of lines of code in the testcase, as computed by sloc [2]. Trace is the average maximum trace length computed by the CodeSurfer plugin. The next four columns show the average number of predicates generated by SatAbs to check all relevant claims in the testcase at buffer sizes one through four. Tests which SatAbs failed to check successfully are denoted by a dash.

| Program | Vulnerability | Testcase | Safe | | | | | | Unsafe | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | Buffer size | | | | | | Buffer size | | | |
| | | | LOC | Trace | 1 | 2 | 3 | 4 | LOC | Trace | 1 | 2 | 3 | 4 |
| Apache 1 | CVE-2004-0940 | iter1_prefixLong_arr | 83 | 25.25 | 40.25 | 45.75 | 49.5 | 53.25 | 77 | 25 | 52.67 | 59.67 | 62.33 | 68.33 |
| Apache 1 | CVE-2004-0940 | iter1_prefixShort_arr | 46 | 24.25 | 29.75 | 35.25 | 39 | 42.75 | 40 | 22.67 | 39.33 | 43.67 | 49.33 | 52 |
| Apache 2 | CVE-2006-3747 | simp1 | 27 | 8 | 5 | 5 | 5 | 5 | 27 | 8 | 4 | 10 | 18 | 27 |
| Apache 2 | CVE-2006-3747 | simp2 | 33 | 17 | 59 | 59 | 59 | 59 | 33 | 17 | 73 | 96 | 115 | 134 |
| Apache 2 | CVE-2006-3747 | simp3 | 42 | 24 | 71 | 71 | 71 | 71 | 42 | 24 | 89 | 113 | 132 | 151 |
| Apache 2 | CVE-2006-3747 | strncmp | 32 | | 37 | 37 | 37 | 37 | 32 | | 36 | 42 | 50 | 59 |
| Apache 2 | CVE-2006-3747 | full | 45 | 24 | 107 | 107 | 107 | 107 | 45 | 24 | — | — | — | — |
| Apache 2 | CVE-2006-3747 | full_ptr | 44 | 24 | 130 | 130 | 130 | 130 | 44 | 24 | — | — | — | — |
| Bind 1 | CA-1999-14 | sig-both | 48 | 27 | 13 | 13 | 13 | 78 | 48 | 27 | — | — | — | — |
| Bind 1 | CA-1999-14 | sig-expand | 45 | 25 | 13 | 13 | 13 | 78 | 45 | 25 | — | — | — | — |
| Bind 2 | CVE-2001-0011 | small | 56 | 0.67 | 11 | 11 | 11 | 11 | 56 | 1 | 9 | 11 | 16 | 24 |
| Bind 2 | CVE-2001-0011 | med | 90 | 0.67 | 14 | 14 | 14 | 14 | 90 | 1 | 12 | 14 | 19 | 27 |
| Bind 2 | CVE-2001-0011 | big | 191 | 1.33 | 41 | 41 | 41 | 41 | 191 | 2 | 39 | 41 | 46 | 54 |
| Edbrowse | CVE-2006-6909 | no_strcmp | 49 | 5 | — | — | — | — | 49 | 7.5 | 55 | 57 | 62 | 73 |
| Edbrowse | CVE-2006-6909 | strchr | 19 | 5 | — | — | — | — | 19 | 7.5 | 30 | 32 | 37 | 48 |
| LibGD | CVE-2007-0455 | gd_no_entities | 87 | 4 | 41 | 62 | 80 | 95 | 86 | 4 | 14 | 28 | 40 | 48 |
| LibGD | CVE-2007-0455 | gd_simp | 71 | 4 | 21 | 31 | 41 | 51 | 70 | 4 | 19 | 29 | 39 | 49 |
| LibGD | CVE-2007-0455 | gd_full | 151 | 4 | — | — | — | — | 150 | 4 | 14 | 20 | 37 | 73 |
| MADWiFi | CVE-2006-6332 | no_sprintf | 32 | 10 | 28 | 28 | 59 | 59 | 31 | 8 | — | — | — | — |
| MADWiFi | CVE-2006-6332 | interproc | 44 | 10 | 28 | 59 | 59 | 92 | 43 | 8 | — | — | — | — |
| NetBSD | CVE-2006-6652 | glob1-bounds | 8 | 1 | 2 | 2 | 2 | 2 | 8 | 1 | 2 | 2 | 2 | 2 |
| NetBSD | CVE-2006-6652 | glob2-int | 57 | 20.25 | 2 | 2 | 2 | 2 | 57 | 20.25 | — | — | — | — |
| NetBSD | CVE-2006-6652 | glob2-anyMeta_int | 34 | 11 | 26 | 47 | 64 | 84 | 34 | 11 | 11 | 19 | 30 | 42 |
| NetBSD | CVE-2006-6652 | glob2-noAnyMeta_int | 31 | 8 | 22 | 41 | 56 | 74 | 31 | 8 | 9 | 16 | 26 | 37 |
| NetBSD | CVE-2006-6652 | glob2-loop | 16 | 5 | 38 | 61 | 80 | 99 | 16 | 5 | 26 | 47 | 70 | 89 |
| NetBSD | CVE-2006-6652 | glob3-int | 75 | 19.33 | 2 | 2 | 2 | 2 | 75 | 19.33 | 4 | 11 | 15 | 19 |
| OpenSER | CVE-2006-6749 | cases1_stripNone_arr | 32 | 6.67 | 20 | 19 | 19 | 19 | 29 | 6 | 10 | 19 | 19 | 37 |
| OpenSER | CVE-2006-6749 | cases1_stripSpacesStart_arr | 33 | 8 | 22 | 20 | 20 | 20 | 30 | 7.33 | 11 | 28 | 45 | 45 |
| OpenSER | CVE-2006-6749 | cases1_stripSpacesEnd_arr | 33 | 8.67 | 28 | 27 | 27 | 27 | 30 | 8 | 16 | 27 | 48 | 48 |
| OpenSER | CVE-2006-6749 | cases1_stripSpacesBoth_arr | 34 | 10 | 30 | 28 | 28 | 28 | 31 | 9.33 | — | — | — | — |
| OpenSER | CVE-2006-6749 | cases1_stripFullStart_arr | 34 | 10 | 26 | 22 | 22 | 22 | 31 | 9.33 | 13 | 26 | 55 | 55 |
| OpenSER | CVE-2006-6749 | cases1_stripFullEnd_arr | 34 | 11.33 | 36 | 35 | 35 | 35 | 31 | 10.67 | 22 | 35 | 64 | 64 |
| OpenSER | CVE-2006-6749 | cases1_stripFullBoth_arr | 36 | 14.67 | 42 | 38 | 38 | 38 | 33 | 14 | — | — | — | — |
| OpenSER | CVE-2006-6749 | cases2_stripNone_arr | 33 | 6.67 | 25 | 24 | 24 | 24 | 30 | 6 | 13 | 24 | 24 | 54 |
| OpenSER | CVE-2006-6749 | cases2_stripSpacesStart_arr | 34 | 8 | 27 | 25 | 25 | 25 | 31 | 7.33 | 14 | 33 | 52 | 61 |
| OpenSER | CVE-2006-6749 | cases2_stripSpacesEnd_arr | 34 | 8.67 | 33 | 32 | 32 | 32 | 31 | 8 | 19 | 32 | 55 | 55 |
| OpenSER | CVE-2006-6749 | cases2_stripSpacesBoth_arr | 35 | 10 | 35 | 33 | 33 | 33 | 32 | 9.33 | — | — | — | — |
| OpenSER | CVE-2006-6749 | cases2_stripFullStart_arr | 35 | 10 | 31 | 27 | 27 | 27 | 32 | 9.33 | 16 | 43 | 62 | 62 |
| OpenSER | CVE-2006-6749 | cases2_stripFullEnd_arr | 35 | 11.33 | 41 | 40 | 40 | 40 | 32 | 10.67 | 25 | 40 | 71 | 71 |
| OpenSER | CVE-2006-6749 | cases2_stripFullBoth_arr | 37 | 14.67 | 47 | 43 | 43 | 43 | 34 | 14 | — | — | — | — |

| | | | Safe | | | | | | Unsafe | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | Buffer size | | | | | | Buffer size | | | |
| Program | Vulnerability | Testcase | LOC | Trace | 1 | 2 | 3 | 4 | LOC | Trace | 1 | 2 | 3 | 4 |
| OpenSER | CVE-2006-6749 | cases3_stripNone_arr | 35 | 8.67 | 27 | 31 | 26 | 26 | 32 | 8 | 21 | 31 | 31 | 57 |
| OpenSER | CVE-2006-6749 | cases3_stripSpacesStart_arr | 36 | 10 | 39 | 39 | 39 | 39 | 33 | 9.33 | 22 | 43 | 41 | 60 |
| OpenSER | CVE-2006-6749 | cases3_stripSpacesEnd_arr | 36 | 10.67 | 40 | 32 | 39 | 39 | 33 | 10 | 19 | 32 | 32 | 92 |
| OpenSER | CVE-2006-6749 | cases3_stripSpacesBoth_arr | 37 | 12 | 39 | 39 | 39 | 39 | 34 | 11.33 | 34 | 47 | 54 | 99 |
| OpenSER | CVE-2006-6749 | cases3_stripFullStart_arr | 37 | 12 | 39 | 35 | 35 | 39 | 34 | 11.33 | 23 | 58 | 70 | 70 |
| OpenSER | CVE-2006-6749 | cases3_stripFullEnd_arr | 37 | 13.33 | 39 | 38 | 43 | 38 | 34 | 12.67 | 25 | 70 | 84 | 84 |
| OpenSER | CVE-2006-6749 | cases3_stripFullBoth_arr | 39 | 16.67 | 61 | 64 | 64 | 60 | 36 | 16 | — | — | — | — |
| Sendmail 1 | CVE-1999-0047 | arr_one_char_no_test | 22 | 6.5 | 5 | 6 | 6 | 6 | 20 | 4.5 | 4 | 8 | 10 | 12 |
| Sendmail 1 | CVE-1999-0047 | arr_one_char_med_test | 33 | 11.75 | 9 | 9 | 9 | 9 | 33 | 11 | 9 | 13 | 15 | 17 |
| Sendmail 1 | CVE-1999-0047 | arr_one_char_heavy_test | 35 | 23 | 13 | 13 | 13 | 13 | 35 | 23 | 15 | 19 | 21 | 23 |
| Sendmail 1 | CVE-1999-0047 | ptr_one_char_no_test | 21 | 6.5 | 10 | 19 | 19 | 19 | 19 | 4.5 | 2 | 2 | 2 | 2 |
| Sendmail 1 | CVE-1999-0047 | ptr_one_char_med_test | 30 | 14.25 | 18.67 | 22.33 | 22.33 | 22.33 | 30 | 13.25 | — | — | — | — |
| Sendmail 1 | CVE-1999-0047 | ptr_one_char_heavy_test | 32 | 24.5 | 24.67 | 28.33 | 28.33 | 28.33 | 32 | 24.5 | — | — | — | — |
| Sendmail 1 | CVE-1999-0047 | arr_two_chars_no_test | 29 | 12.33 | 6 | 7.67 | 8.33 | 8.33 | 25 | 8.33 | — | — | — | — |
| Sendmail 1 | CVE-1999-0047 | arr_two_chars_med_test | 49 | 22.29 | 11.17 | 11.5 | 12.33 | 11.5 | 49 | 21.29 | 7 | 12.5 | 16 | 19.5 |
| Sendmail 1 | CVE-1999-0047 | arr_two_chars_heavy_test | 54 | 32 | 17.83 | 18.17 | 18.17 | 18.17 | 54 | 31.14 | 12 | 22.5 | 26 | 29.5 |
| Sendmail 1 | CVE-1999-0047 | ptr_two_chars_no_test | 27 | 12.33 | 11 | 14 | 19.5 | 30.5 | 23 | 8.33 | 2 | 2 | 2 | 2 |
| Sendmail 1 | CVE-1999-0047 | ptr_two_chars_med_test | 43 | 22.29 | 19 | 22.5 | 25.75 | 25.75 | 43 | 21.14 | — | — | — | — |
| Sendmail 1 | CVE-1999-0047 | ptr_two_chars_heavy_test | 48 | 30.5 | 24 | 28.33 | 32.67 | 32.67 | 48 | 30.5 | — | — | — | — |
| Sendmail 1 | CVE-1999-0047 | arr_three_chars_no_test | 36 | 18 | 7 | 8.33 | 9 | 10 | 30 | 12.25 | — | — | — | — |
| Sendmail 1 | CVE-1999-0047 | arr_three_chars_med_test | 67 | 26.9 | 11.29 | 11.86 | 12.57 | 11.86 | 67 | 25.5 | 15 | 12 | 20 | 23 |
| Sendmail 1 | CVE-1999-0047 | arr_three_chars_heavy_test | 75 | 46.4 | 19.29 | 21.14 | 21.14 | 20.14 | 75 | 45 | 31 | 28 | 36 | 39 |
| Sendmail 1 | CVE-1999-0047 | ptr_three_chars_no_test | 33 | 18 | 12 | 14.67 | 18 | 27 | 27 | 12.25 | 2 | 2 | 2 | 2 |
| Sendmail 1 | CVE-1999-0047 | ptr_three_chars_med_test | 58 | 27.7 | 19 | 22.6 | 27.4 | 27.4 | 58 | 26.4 | — | — | — | — |
| Sendmail 1 | CVE-1999-0047 | ptr_three_chars_heavy_test | 66 | 40.22 | 29 | 33.25 | 37 | 37 | 66 | 40.89 | — | — | — | — |
| Sendmail 2 | CVE-2002-1337 | close-angle_ptr_no_test | 29 | 8 | 11 | 25 | 44 | 57 | 29 | 8 | 8 | 22 | 33 | 44 |
| Sendmail 2 | CVE-2002-1337 | close-angle_ptr_one_test | 39 | 11 | 12 | 28 | 50 | 65 | 39 | 11 | 9 | 25 | 39 | 52 |
| WU-ftpd | CVE-1999-0368 | curpath-simple | 26 | 0.67 | — | — | — | — | 26 | 1 | 17 | 21 | 25 | 29 |
| WU-ftpd | CVE-1999-0368 | namebuf-iter_ints_simp | 61 | 0 | 26 | 26 | 26 | 26 | 62 | 9.75 | — | — | — | — |
| WU-ftpd | CVE-1999-0368 | namebuf-iter_ints | 82 | 0 | 77 | 77 | 77 | 77 | 82 | 4.5 | — | — | — | — |
| WU-ftpd | CVE-1999-0368 | linkpath-strcpy_strcat | 43 | 0 | 46.5 | 52.5 | 57.5 | 62.5 | 37 | 0.17 | 29 | 34.5 | 39 | 43.5 |
| WU-ftpd | CVE-1999-0368 | namebuf-strcpy_strcat | 30 | 0 | 75 | 94 | 113 | 126 | 30 | 0 | 52 | 59 | 71 | 78 |

# Appendix B

# Codesurfer plugin source code

The source code of the CodeSurfer plugin, written in the STk dialect of Scheme, is included below. As described in Section 2.4.4, given an SDG (automatically computed by Codesurfer for a given program) the plugin computes the maximum trace length for each statement labeled "VULN" and outputs the average maximum length. The main statement is at the bottom; each major function is preceded by a comment (denoted by the ";" character) containing a brief description.

```
1  (define (pdg-vertex->pdg-vertex-set v)
2   (let ((vset (pdg-vertex-set-create)))
3    (pdg-vertex-set-put! vset v)
4    vset))
5
6  ; Returns the possibly empty label of pdg-vertex vertex.
7  (define (pdg-vertex-label vertex)
8   (let ((str-out ""))
9   (pdg-vertex-set-traverse
10   (cfg-edge-set->pdg-vertex-set (pdg-vertex-cfg-predecessors vertex))
11   (lambda (v)
12    (if (and (eq? (pdg-vertex-kind v) 'label) (pdg-vertex-characters v))
13     (set! str-out (pdg-vertex-characters v)))))
14  str-out))
15
16 (define (print-pdg-vertex vertex)
17  (begin
18    (format #t "~a " (pdg-vertex-characters vertex))
19    (format #t " <~a>" (pdg-vertex-label vertex))))
20
21 (define (print-pdg-vertex-set vset)
22  (pdg-vertex-set-traverse
23   vset
24   (lambda (v)
25    (print-pdg-vertex v)
26    (display " "))))
```

```
27
28 (define (print-pdg-vertex-list l)
29  (for-each
30   (lambda (pv)
31    (print-pdg-vertex pv)
32    (display ", "))
33   l))
34
35 (define (csucc vset)
36   (s-successors vset '(control ())))
37
38 (define (cpred vset)
39   (s-predecessors vset '(control ())))
40
41 (define (dsucc vset)
42   (s-successors vset '(data ())))
43
44 (define (dpred vset)
45   (s-predecessors vset '(data ())))
46
47 ; Returns control and data predecessors of all vertices in vset.
48 (define (cdpred vset)
49  (pdg-vertex-set-union
50   (dpred vset)
51   (cpred vset)))
52
53 ; Returns the result of applying fn to each pdg-vertex in pvset, filtering out
54 ; vertices in pvset and non-expression vertices. Function fn is pdg-vertex-set
55 ; -> pdg-vertex-set.
56 (define (img fn pvset)
57  (let ((pvset-out (pdg-vertex-set-create)))
58   (pdg-vertex-set-traverse
59    (fn pvset)
60    (lambda (pv)
61     (if
62      (and
63       (let ((kind (pdg-vertex-kind pv)))
64        (or
65         (eq? kind 'expression)
66         (eq? kind 'control-point)
67         (eq? kind 'jump)
68         (eq? kind 'formal-out)
69         (eq? kind 'formal-in)
70         (eq? kind 'actual-out)
71         (eq? kind 'actual-in)))
72       (not (pdg-vertex-set-member? pvset pv)))
73      (begin
74       (pdg-vertex-set-put! pvset-out pv)
75       ))))
76   pvset-out))
77
78 (define (cfg-edge-set->pdg-vertex-set ceset)
79  (let ((pvset (pdg-vertex-set-create)))
80   (cfg-edge-set-traverse ceset
81    (lambda (pv el)
82     (pdg-vertex-set-put! pvset pv)))
83   pvset))
84
85 (define (pdg-vertex-set-cfg-predecessors pvset-in)
86  (let ((pvset-out (pdg-vertex-set-create)))
87   (pdg-vertex-set-traverse pvset-in
88    (lambda (pv)
89     (set! pvset-out
90      (pdg-vertex-set-union pvset-out
91       (cfg-edge-set->pdg-vertex-set
92        (pdg-vertex-cfg-predecessors pv))))
93     ))
94   pvset-out))
```

```
95
96  (define (actual-ins callsite-pv)
97   (let ((out-set (pdg-vertex-set-create)))
98    (pdg-vertex-set-traverse
99     (csucc (pdg-vertex->pdg-vertex-set callsite-pv))
100    (lambda (v)
101     (if (eq? (pdg-vertex-kind v) 'actual-in)
102      (pdg-vertex-set-put! out-set v))))
103   out-set))
104
105 ; Returns the possibly empty pdg-vertex-set of all pdg-vertices with label
106 ; containing label as a substring.
107 (define (labelled-pdg-vertices label)
108  (let ((out-set (pdg-vertex-set-create)))
109   (for-each
110    (lambda (pdg)
111     (pdg-vertex-set-traverse
112      (pdg-vertices pdg)
113      (lambda (v)
114       (if (string-find? label (pdg-vertex-label v))
115        (if (eq? (pdg-vertex-kind v) 'call-site)
116         (set! out-set (pdg-vertex-set-union out-set (actual-ins v)))
117         (pdg-vertex-set-put! out-set v))))))
118    (sdg-pdgs))
119   out-set))
120
121 (define (list-prefix l n)
122  (if (or (eq? n 0) (null? l))
123   '()
124   (cons (car l) (list-prefix (cdr l) (- n 1)))))
125
126 ; Returns length of longest suffix of path, using fn as the successor function.
127 ; Search stops on a branch if a cycle is detected.
128 ; Stops if MAX-PATHS paths have been explored.
129 (define (dfs fn path)
130  (if (< num-paths MAX-PATHS)
131   (let* ((tail (car (reverse path)))
132          (image (img fn (pdg-vertex->pdg-vertex-set tail)))
133          (filtered-image
134           (let ((out-set (pdg-vertex-set-create)))
135            (pdg-vertex-set-traverse
136             image
137             (lambda (v)
138              (if (not (member v path))
139               (pdg-vertex-set-put! out-set v))))
140            out-set))
141          (max-suffix-len 0))
142    (for-each
143     (lambda (v)
144       ; tail->v doesn't form a cycle--got an extension
145       (let ((new-path (append path (list v))))
146        (set! max-suffix-len
147         (max max-suffix-len (+ 1 (dfs fn new-path))))))
148      (list-prefix (pdg-vertex-set->list filtered-image) MAX-BRANCH))
149
150     (if (eq? max-suffix-len 0) ; reached a leaf/cycle
151      (begin
152       (set! num-paths (+ num-paths 1))
153       ))
154      max-suffix-len)
155   0)) ; empty path
156
157 (define (max-length ll)
158  (if (null? ll)
159   0
160   (max (length (car ll)) (max-length (cdr ll)))))
161
162 (define num-paths 0)
```

```
163
164 ; MAIN STATEMENT. Calls dfs on each pdg vertex labelled "VULN". Prints results
165 ; to stdout.
166 (let ((max-length-sum 0)
167       (path-sets 0))
168   (pdg-vertex-set-traverse
169    (labelled-pdg-vertices "VULN")
170    (lambda (v)
171     (set! num-paths 0)
172     (set! max-length-sum (+ max-length-sum (dfs cdpred (list v))))
173     (set! path-sets (+ path-sets 1))))
174   (format #t "#pathsets: ~a\n" path-sets)
175   (format #t "#paths: ~a\n" num-paths)
176   (format #t "max length sum: ~a\n" max-length-sum)
177   (format #t "avg. max length: ~a\n" (/ max-length-sum path-sets)))
178
179 (quit)
```