View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by DSpace@MIT

Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2013-018 August 6, 2013

Sound Input Filter Generation for Integer Overflow Errors

Fan Long, Stelios Sidiroglou-Douskos, Deokhwan Kim, and Martin Rinard

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Sound Input Filter Generation for Integer Overflow Errors

Fan Long, Stelios Sidiroglou-Douskos, Deokhwan Kim, Martin Rinard
MIT CSAIL
{fanl, stelios, dkim, rinard}@csail.mit.edu

Abstract

We present a system, SIFT, for generating input filters that nullify integer overflow errors associated with critical program sites such as memory allocation or block copy sites. SIFT uses a static program analysis to generate filters that discard inputs that may trigger integer overflow errors in the computations of the sizes of allocated memory blocks or the number of copied bytes in block copy operations. The generated filters are sound — if an input passes the filter, it will not trigger an integer overflow error for any analyzed site.

Our results show that SIFT successfully analyzes (and therefore generates sound input filters for) 52 out of 58 memory allocation and block memory copy sites in analyzed input processing modules from five applications (VLC, Dillo, Swfdec, Swftools, and GIMP). These nullified errors include six known integer overflow vulnerabilities. Our results also show that applying these filters to 62895 real-world inputs produces no false positives. The analysis and filter generation times are all less than a second.

1. Introduction

Many security exploits target software errors in deployed applications. One approach to nullifying vulnerabilities is to deploy input filters that discard inputs that may trigger the errors.

We present a new static analysis technique and implemented system, SIFT, for automatically generating filters that discard inputs that may trigger integer overflow errors at analyzed memory allocation and block copy sites. We focus on this problem, in part, because of its practical importance. Because integer overflows may enable code injection or other attacks, they are an important source of security vulnerabilities [22, 29, 32].

Unlike all previous techniques of which we are aware, SIFT is *sound* — if an input passes the filter, it will not trigger an overflow error at any analyzed site.

1.1 Previous Filter Generation Systems

Standard filter generation systems start with an input that triggers an error [8–10, 24, 33]. They next use the input to generate an execution trace and discover the path the program takes to the error. They then use a forward symbolic execution on the discovered path (and, in some cases, heuristically related paths) to derive a *vulnerability signature* — a boolean condition that the input must satisfy to follow the same execution path through the program to trigger the same error. The generated filter discards inputs that satisfy the vulnerability signature. Because other unconsidered paths to the error may exist, these techniques are unsound (i.e., the filter may miss inputs that exploit the error).

It is also possible to start with a potentially vulnerable site and use a weakest precondition analysis to obtain an input filter for that site. To our knowledge, the only previous technique that uses this approach [4] is unsound in that 1) it uses loop unrolling to eliminate loops and therefore analyzes only a subset of the possible execution paths and 2) it does not specify a technique for dealing with potentially aliased values. As is standard, the generated filter incorporates checks from conditional statements along the analyzed execution paths. The goal is to avoid filtering potentially problematic inputs that the program would (because of safety checks at conditionals along the execution path) process correctly. As a result, the generated input filters perform a substantial (between 10^6 and 10^{10}) number of operations.

1.2 SIFT

SIFT starts with a set of *critical expressions* from memory allocation and block copy sites. These expressions control the sizes of allocated or copied memory blocks at these sites. SIFT then uses an interprocedural, demand-driven, weakest precondition static analysis to propagate the critical expression backwards against the control flow. The result is a symbolic condition that captures all expressions that the application may evaluate (in any execution) to obtain the values of critical expressions. The free variables in the symbolic condition represent the values of input fields. In effect, the symbolic condition captures all of the possible computations that the program may perform on the input fields to obtain the values of critical expressions. Given an input, the generated input filter evaluates this condition over the corresponding input fields to discard inputs that may cause an overflow.

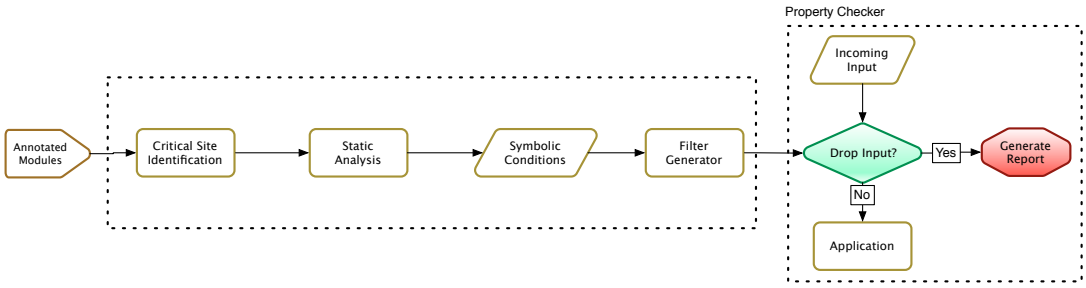

Figure 1. The SIFT architecture.

A key challenge is that, to successfully extract effective symbolic conditions, the analysis must reason precisely about interprocedural computations that use pointers to compute and manipulate values derived from input fields. Our analysis meets this challenge by deploying a novel combination of techniques including 1) a novel interprocedural weakest precondition analysis that works with symbolic representations of input fields and values accessed via pointers (including input fields read in loops and values accessed via pointers in loops) and 2) an alias analysis that ensures that the derived symbolic condition correctly characterizes the values that the program computes.

Another key challenge is obtaining loop invariants that enable the analysis to precisely characterize how loops transform the propagated symbolic conditions. Our analysis meets this challenge with a novel symbolic expression normalization algorithm that enables the fixed point analysis to terminate unless it attempts to compute a symbolic value that depends on a statically unbounded number of values computed in different loop iterations (see Section 3.2).

- **Sound Filters:** Because SIFT takes all paths to analyzed memory allocation and block copy sites into account, it generates sound filters — if an input passes the filter, it will not trigger an overflow in the evaluation of any critical expression (including the evaluation of intermediate expressions at distant program points that contribute to the value of the critical expression).[1]
- **Efficient Filters:** Unlike standard techniques, SIFT incorporates no checks from the program's conditional statements and works only with arithmetic expressions that contribute directly to the values of the critical expressions. It therefore generates much more efficient filters than standard techniques (SIFT's filters perform tens of operations as opposed to tens of thousands or more). Indeed, our experimental results show that, in contrast to standard filters, SIFT's filters spend essentially all of their time reading the input (as opposed to checking if the input may trigger an overflow error).
- **Accurate Filters:** Our experimental results show that, empirically, ignoring execution path constraints results in no loss of accuracy. Specifically, we tested our generated filters on 62895 real-world inputs for six benchmark applications and found no false positives (incorrectly filtered inputs that the program would have processed correctly). We attribute this potentially counterintuitive result to the fact that standard integer data types usually contain enough bits to represent the memory allocation sizes and block copy lengths that benign inputs typically elicit.

[1] As is standard in the field, SIFT uses an alias analysis that is designed to work with programs that do not access uninitialized or out of bounds memory. Our analysis therefore comes with the following soundness guarantee. If an input passes the filter for a given critical expression e, the input field annotations are correct (see Section 3.4), and the program has not yet accessed uninitialized or out of bounds memory when the program computes a value of e, then no integer overflow occurs during the evaluation of e (including the evaluations of intermediate expressions that contribute to the final value of the critical expression).

1.3 SIFT Usage Model

Figure 1 presents the architecture of SIFT. The architecture is designed to support the following usage model:

Module Identification. Starting with an application that is designed to process inputs presented in one or more input formats, the developer identifies the modules within the application that process inputs of interest. SIFT will analyze these modules to generate an input filter for the inputs that these modules process.

Input Statement Annotation. The developer annotates the relevant input statements in the source code of the modules to identify the input field that each input statement reads.

Critical Site Identification. SIFT scans the modules to find all *critical sites* (currently, memory allocation and block copy sites). Each critical site has a *critical expression* that determines the size of the allocated or copied block of memory. The generated input filter will discard inputs that may trigger an integer overflow error during the computation of the value of the critical expression.

Static Analysis. For each critical expression, SIFT uses a demand-driven backwards static program analysis to auto-

matically derive the corresponding *symbolic condition*. Each conjunct expression in this condition specifies, as a function of the input fields, how the value of the critical expression is computed along one of the program paths to the corresponding critical site.

Input Parser Acquisition. The developer obtains (typically from open-source repositories such as Hachoir [1]) a parser for the desired input format. This parser groups the input bit stream into input fields, then makes these fields available via a standard API.

Filter Generation. SIFT uses the input parser and symbolic conditions to automatically generate the input filter. When presented with an input, the filter reads the fields of the input and, for each symbolic expression in the conditions, determines if an integer overflow may occur when the expression is evaluated. If so, the filter discards the input. Otherwise, it passes the input along to the application. The generated filters can be deployed anywhere along the path from the input source to the application that ultimately processes the input.

1.4 Experimental Results

We used SIFT to generate input filters for modules in five real-world applications: VLC 0.8.6h (a network media player), Dillo 2.1 (a lightweight web browser), Swfdec 0.5.5 (a flash video player), Swftools 0.9.1 (SWF manipulation and generation utilities), and GIMP 2.8.0 (an image manipulation application). Together, the analyzed modules contain 58 critical memory allocation and block copy sites. SIFT successfully generated filters for 52 of these 58 critical sites (SIFT's static analysis was unable to derive symbolic conditions for the remaining six critical sites, see Section 5.2 for more details). These applications contain six integer overflow vulnerabilities at their critical sites. SIFT's filters nullify all of these vulnerabilities.

Analysis and Filter Generation Time. We configured SIFT to analyze all critical sites in the analyzed modules, then generate a single, high-performance composite filter that checks for integer overflow errors at all of the sites. The maximum time required to analyze all of the sites and generate the composite filter was less than a second for each benchmark application.

False Positive Evaluation. We used a web crawler to obtain a set of least 6000 real-world inputs for each application (for a total of 62895 input files). We found no false positives — the corresponding composite filters accept all of the input files in this test set.

Filter Performance. We measured the composite filter execution time for each of the 62895 input files in our test set. The average time required to read and filter each input was at most 16 milliseconds, with this time dominated by the time required to read in the input file.

1.5 Contributions

This paper makes the following contributions:

- **SIFT:** We present SIFT, a sound filter generation system for nullifying integer overflow vulnerabilities. SIFT scans modules to find critical memory allocation and block copy sites, statically analyzes the code to automatically derive symbolic conditions that characterize how the application may compute the sizes of the allocated or copied memory blocks, and generates input filters that discard inputs that may trigger integer overflow errors in the evaluation of these expressions.

  In comparison with previous filter generation techniques, SIFT is sound and generates efficient and empirically precise filters.
- **Static Analysis:** We present a new static analysis that automatically derives symbolic conditions that capture, as a function of the input fields, how the integer values of critical expressions are computed along the various possible execution paths to the corresponding critical site.

  Key elements of this static analysis include 1) a precise backwards symbolic analysis that soundly and accurately reasons about symbolic conditions in the face of instructions that use pointers to load and store computed values and 2) a novel normalization procedure that enables the analysis to effectively synthesize symbolic loop invariants.
- **Experimental Results:** We present experimental results that illustrate the practical viability of our approach in protecting applications against integer overflow vulnerabilities at memory allocation and block copy sites.

The remainder of the paper is organized as follows. Section 2 presents an example that illustrates how SIFT works. Section 3 presents the core SIFT static analysis for C programs. Section 4 presents the formalization of the static analysis and discusses the soundness of the analysis. Section 5 presents the experimental results. Section 6 presents related work. We conclude in Section 7.

2. Example

We next present an example that illustrates how SIFT nullifies an integer overflow vulnerability in Swfdec 0.5.5, an open source shockwave flash player.

Figure 2 presents (simplified) source code from Swfdec. When Swfdec opens an SWF file with embedded JPEG images, it calls $\mathrm{jpeg\_decoder\_decode}()$ (line 1 in Figure 2) to decode each JPEG image in the file. This function in turn calls the function $\mathrm{jpeg\_decoder\_start\_of\_frame}()$ (line 7) to read the image metadata and the function

```
1  int jpeg_decoder_decode(JpegDecoder *dec) {
2    ...
3    jpeg_decoder_start_of_frame(dec, ...);
4    jpeg_decoder_init_decoder (dec);
5    ...
6  }
7  void jpeg_decoder_start_of_frame(JpegDecoder*dec){
8    ...
9    dec->height = jpeg_bits_get_u16_be (bits);
10   /* dec->height = SIFT_input("jpeg_height", 16);*/
11   dec->width = jpeg_bits_get_u16_be (bits);
12   /* dec->width = SIFT_input("jpeg_width", 16); */
13   for (i = 0; i < dec->n_components; i++) {
14     dec->components[i].h_sample =getbits(bits, 4);
15     /* dec->components[i].h_sample =
16          SIFT_input("h_sample", 4); */
17     dec->components[i].v_sample =getbits(bits, 4);
18     /* dec->components[i].v_sample =
19          SIFT_input("v_sample", 4); */
20   }
21 }
22 void jpeg_decoder_init_decoder(JpegDecoder*dec){
23   int max_h_sample = 0;
24   int max_v_sample = 0;
25   int i;
26   for (i=0; i < dec->n_components; i++) {
27     max_h_sample = MAX(max_h_sample,
28       dec->components[i].h_sample);
29     max_v_sample = MAX(max_v_sample,
30       dec->components[i].v_sample);
31   }
32   dec->width_blocks=(dec->width+8*max_h_sample-1)
33     / (8*max_h_sample);
34   dec->height_blocks=(dec->height+8*max_v_sample-1)
35     / (8*max_v_sample);
36   for (i = 0; i < dec->n_components; i++) {
37     int rowstride;
38     int image_size;
39     dec->components[i].h_subsample=max_h_sample /
40      dec->components[i].h_sample;
41     dec->components[i].v_subsample=max_v_sample /
42      dec->components[i].v_sample;
43     rowstride=dec->width_blocks * 8 * max_h_sample /
44      dec->components[i].h_subsample;
45     image_size=rowstride * (dec->height_blocks * 8 *
46      max_v_sample / dec->components[i].v_subsample);
47     dec->components[i].image = malloc (image_size);
48   }
49 }
```

Figure 2. Simplified Swfdec source code. Input statement annotations appear in comments.

jpeg_decoder_init_decoder() (line 22) to allocate memory buffers for the JPEG image.

There is an integer overflow vulnerability at lines 43-47 where Swfdec calculates the size of the buffer for a JPEG image as:

```
rowstride * (dec->height_block * 8 * max_v_sample /
dec->components[i].v_subsample)
```

At this program point, rowstride equals:

```
(jpeg_width + 8 * max_h_sample - 1) / (8 * max_h_sample)
* 8 * max_h_sample / (max_h_sample / h_sample)
```

while the rest of the expression equals

```
(jpeg_height + 8 * max_v_sample - 1) / (8 * max_v_sample)
* 8 * max_v_sample / (max_v_sample / v_sample)
```

where jpeg_height is the 16-bit height input field value that Swfdec reads at line 9 and jpeg_width is the 16-bit width input field value that Swfdec reads at line 11. h_sample is one of the horizontal sampling factor values that Swfdec reads at line 14, while max_h_sample is the maximum horizontal sampling factor value. v_sample is one of the vertical sampling factor values that Swfdec reads at line 17, while max_v_sample is the maximum vertical sampling factor value. Malicious inputs with specifically crafted values in these input fields can cause the image buffer size calculation to overflow. In this case Swfdec allocates an image buffer that is smaller than required and eventually writes beyond the end of the allocated buffer.

The loop at lines 13-20 reads an array of horizontal and vertical factor values. Swfdec computes the maximum values of these factors in the loop at lines 26-31. It then uses these values to compute the size of the allocated buffer at each iteration in the loop (lines 36-48).

Analysis Challenges: This example highlights several challenges that SIFT must overcome to successfully analyze and generate a filter for this program. First, the expression for the size of the buffer uses pointers to access values derived from input fields. To overcome this challenge, SIFT uses an alias analysis [17] to reason precisely about expressions with pointers.

Second, the memory allocation site (line 47) occurs in a loop, with the size expression referencing input values read in a different loop (lines 13-19). Different instances of the same input field (h_sample and v_sample) are used to compute (potentially different) sizes for different blocks of memory allocated at the same site. To reason precisely about these different instances, the analysis works with an abstraction that materializes, on demand, abstract representatives of accessed input field and computed values (see Section 3). To successfully analyze the loop, the analysis uses a new loop invariant synthesis algorithm (which exploits a new expression normalization technique to reach a fixed point).

Finally, Swfdec reads the input fields (lines 14 and 17) and computes the size of the allocated memory block (lines 45-46) in different procedures. SIFT therefore uses an interprocedural analysis that propagates symbolic conditions across procedure boundaries to obtain precise symbolic conditions.

We next describe how SIFT generates a sound input filter to nullify this integer overflow error.

Source Code Annotations: SIFT provides an declarative specification interface that enables the developer to specify which statements read which input fields. In this example, the developer specifies that the application reads the input fields jpeg_height, jpeg_width, h_sample, and v_sample at lines 10, 12, 15-16, and 18-19 in Figure 2. SIFT uses this specification to map the variables dec->height, dec->width, dec->components[i].h_sample, and dec->components[i].v_sample at lines 9, 11, 14, and

$$C: \quad \text{safe}((((sext(jpeg\_width^{[16]},32)+8^{[32]}\times sext(h\_sample(1)^{[4]},32)-1^{[32]})/(8^{[32]}\times sext(h\_sample(1)^{[4]},32))$$
$$\times 8^{[32]}\times sext(h\_sample(1)^{[4]},32))/(sext(h\_sample(1)^{[4]},32)/sext(h\_sample(2)^{[4]},32)))$$
$$\times(((sext(jpeg\_height^{[16]},32)+8^{[32]}\times sext(v\_sample(1)^{[4]},32)-1^{[32]})/(8^{[32]}\times sext(v\_sample(1)^{[4]},32))$$
$$\times 8^{[32]}\times sext(v\_sample(1)^{[4]},32))/(sext(v\_sample(1)^{[4]},32)/sext(v\_sample(2)^{[4]},32))))$$

Figure 3. The symbolic condition C for the Swfdec example. Subexpressions in C are bit vector expressions. The superscript indicates the bit width of each expression atom. "$sext(v, w)$" is the signed extension operation that transforms the value v to the bit width w.

17 to the corresponding input field values. The field names h_sample and v_sample map to two arrays of input fields that Swfdec reads in the loop at lines 14 and 17.

Compute Symbolic Condition: SIFT uses a demand-driven, interprocedural, backward static analysis to compute the symbolic condition C in Figure 3. We use notation "safe(e)" in Figure 3 to denote that overflow errors should not occur in any step of the evaluation of the expression e. Subexpressions in C are in *bit vector expression* form so that the expressions accurately reflect the representation of the numbers inside the computer as fixed-length bit vectors as well as the semantics of arithmetic and logical operations as implemented inside the computer on these bit vectors.

In Figure 3, the superscripts indicate the bit width of each expression atom. $sext(v, w)$ is the signed extension operation that transforms the value v to the bit width w. SIFT also tracks the sign of each arithmetic operation in C. For simplicity, Figure 3 omits this information. SIFT soundly handles the loops that access the input field arrays h_sample and v_sample. The generated C reflects the fact that the variable dec->components[i].h_sample and the variable max_h_sample might be two different elements in the input array h_sample. In C, $h\_sample(1)$ corresponds to max_h_sample and $h\_sample(2)$ corresponds to dec->components[i].h_sample. SIFT handles v_sample similarly.

C includes all intermediate expressions evaluated at lines 32-35 and 39-46. In this example, C contains only a single term in the form of safe(e). However, if there may be multiple execution paths, SIFT generates a symbolic condition C that conjuncts multiple terms in the form of safe(e) to cover all paths.

Generate Input Filter: Starting with the symbolic condition C, SIFT generates an input filter that discards any input that violates C, i.e., for any term safe(e) in C, the input triggers integer overflow errors when evaluating e (including all subexpressions). The generated filter extracts all instances of the input fields jpeg_height, jpeg_width, h_sample, and v_sample (these are the input fields that appear in C) from an incoming input. It then iterates over all combinations of pairs of the input fields h_sample and v_sample to consider all possible bindings of $h\_sample(1)$, $h\_sample(2)$, $v\_sample(1)$, and $v\_sample(2)$ in C. For each binding, it checks the entire evaluation of C (including the evaluation of all subexpressions) for overflow. If there is no overflow in any evaluation, the filter accepts the input, otherwise it rejects the input.

3. Static Analysis

This section presents the static analysis algorithm in SIFT. We have implemented our static analysis for C programs using the LLVM Compiler Infrastructure [2].

3.1 Core Language and Notation

$$s := l\texttt{: } x \texttt{ = read(}f\texttt{)} \mid l\texttt{: } x \texttt{ = } c \mid l\texttt{: } x \texttt{ = } y \mid$$
$$l\texttt{: } x \texttt{ = } y \texttt{ } op \texttt{ } z \mid l\texttt{: } x \texttt{ = } *p \mid l\texttt{: } *p \texttt{ = } x \mid$$
$$l\texttt{: } p \texttt{ = malloc} \mid l\texttt{: skip} \mid s'\texttt{; } s'' \mid$$
$$l\texttt{: if (}x\texttt{) } s' \texttt{ else } s'' \mid l\texttt{: while (}x\texttt{) \{}s'\texttt{\}}$$

$$s,\ s',\ s'' \in \texttt{Statement} \qquad f \in \texttt{InputField}$$
$$x,\ y,\ z,\ p \in \texttt{Var} \qquad c \in \texttt{Int} \qquad l \in \texttt{Label}$$

Figure 4. The Core Programming Language

Figure 4 presents the core language that we use to present the analysis. The language is modeled on a standard lowered program representation in which 1) nested expressions are converted into sequences of statements of the form l: x = y op z (where x, y, and z are either non-aliased variables or automatically generated temporaries) and 2) all accesses to potentially aliased memory locations occur in load or store statements of the form l: x = $*p$ or l:$*p$ = x. Each statement contains a unique label $l \in$ Label.

A statement of the form "l: x = read(f)" reads a value from an input field f. Because the input may contain multiple instances of the field f, different executions of the statement may return different values. For example, the loop at lines 14-17 in Figure 2 reads multiple instances of the h_sample and v_sample input fields.

Labels and Pointer Analysis: Figure 5 presents three utility functions $first$: Statement $\rightarrow$ Label, $last$: Statement $\rightarrow$ Label, and $labels$: Statement $\rightarrow$ Label in our notations. Intuitively, given a statement s, $first$ maps s to the label that corresponds to the first atomic statement inside s; $last$ maps s to the label that corresponds to the last atomic statement inside s; $labels$ maps s to the set of labels that are inside s.

$$first(s) = \begin{cases} first(s') & s = s'; s'' \\ l & \text{otherwise, } l \text{ is the label of } s \end{cases}$$

$$last(s) = \begin{cases} last(s'') & s = s'; s'' \\ l & \text{otherwise, } l \text{ is the label of } s \end{cases}$$

$$labels(s) =$$

$$\begin{cases} labels(s') \cup labels(s'') & s = s'; s'' \\ \{l\} \cup labels(s') & s = \text{while } (x)\ \{s'\} \\ \{l\} \cup labels(s') \cup labels(s'') & s = \text{if } (x)\ s' \text{ else } s'' \\ \{l\} & \text{otherwise, } l \text{ is the label of } s \end{cases}$$

Figure 5. Definitions of $first$, $last$, and $labels$

We use LoadLabel and StoreLabel to denote the set of labels that correspond to load and store statements, respectively. LoadLabel $\subseteq$ Label and StoreLabel $\subseteq$ Label.

Our static analysis uses an underlying pointer analysis [17] to disambiguate aliases at load and store statements. The pointer analysis provides two functions no_alias and $must\_alias$:

$$no\_alias : (\text{StoreLabel} \times \text{LoadLabel}) \rightarrow \text{Bool}$$
$$must\_alias : (\text{StoreLabel} \times \text{LoadLabel}) \rightarrow \text{Bool}$$

We assume that the underlying pointer analysis is sound so that 1) $no\_alias(l_{\text{store}}, l_{\text{load}}) =$ true only if the load statement at label l_{load} will **never** retrieve a value stored by the store statement at label l_{store}; 2) $must\_alias(l_{\text{store}}, l_{\text{load}}) =$ true only if the load statement at label l_{load} will **always** retrieve a value from the last memory location that the store statement at label l_{store} manipulates (see Section 4.2 for a formal definition of the soundness requirements that the alias anlayisis must satisfy).

3.2 Intraprocedural Analysis

Because it works with a lowered representation, our static analysis starts with a variable v at a critical program point. It then propagates v backward against the control flow to the program entry point. In this way the analysis computes a symbolic condition that soundly captures how the program, starting with input field values, may compute the value of v at the critical program point. The generated filters use the analysis results to check whether the input may trigger an integer overflow error in any of these computations.

Condition Syntax:

Figure 7 presents the definition of *symbolic conditions* that our analysis manipulates and propagates. A condition consists of a set of conjuncts in the form of safe(e), which represents that the evaluation of the *symbolic expression* e should not trigger an overflow condition (including all sub-computations in the evaluation, see Section 4.5 for the formal definition of a program state satisfying a condition).

$$C := C \ \wedge\ \text{safe}(e) \mid \text{safe}(e)$$
$$e := e_1\ op\ e_2 \mid atom$$
$$atom := x \mid c \mid f(id) \mid l(id)$$

$$id \in \{1, 2, \dots\} \qquad x \in \text{Var} \qquad c \in \text{Int}$$
$$l \in \text{LoadLabel} \qquad f \in \text{InputField}$$

Figure 7. The Condition Syntax

Symbolic conditions may contain four kinds of atoms: c represents a constant, x represents the variable x, $f(id)$ represents the value of an input field f (the analysis uses the natural number id to distinguish different instances of f), and $l(id)$ represents a value returned by a load statement with the label l (the analysis uses the natural number id to distinguish values loaded at different executions of the load statement).

Analysis Framework: Given a series of statements s, a label l within s ($l \in labels(s)$), and a symbolic condition C at the program point after the corresponding statement with the label l, our demand-driven backwards analysis computes a symbolic condition $F(s, l, C)$. The analysis ensures that if $F(s, l, C)$ holds before executing s, then C will hold whenever the execution reaches the program point after the corresponding statement with the label l (see Section 4.5 for the formal definition).

Given a program s_0 as a series of statements and a variable v at a critical site associated with the label l, our analysis generates the condition $F(s, l, \text{safe}(v))$ to create an input filter that checks whether the input may trigger an integer overflow error in the computations that the program performs to obtain the value of v at the critical site.

Analysis of Assignment, Conditional, and Sequence Statements: Figure 6 presents the analysis rules for basic program statements. The analysis of assignment statements replaces the assigned variable x with the assigned value (c, y, y op z, or $f(id)$, depending on the assignment statement). Here the notation $C[e_a/e_b]$ denotes the new symbolic condition obtained by replacing every occurrence of e_b in C with e_a. The analysis rule for input read statements materializes a new id to represent the read value $f(id)$. This mechanism enables the analysis to correctly distinguish different instances of the same input field (because different instances have different ids).

If the label l identifies the end of a conditional statement, the analysis of the statement takes the union of the symbolic conditions from the analysis of the true and false branches of the conditional statement. The resulting symbolic condition correctly takes the execution of both branches into account. If the label l identifies a program point within one of the branches of a conditional statement, the analysis will propagate the condition from that branch only. The analysis of

Statement s	Rules
$l:\ x = c$	$F(s,l,C) = C[c/x]$
$l:\ x = y$	$F(s,l,C) = C[y/x]$
$l:\ x = y\ op\ z$	$F(s,l,C) = C[y\ op\ z/x]$
$l:\ x = \text{read}(f)$	$F(s,l,C) = C[f(id)/x]$, $f(id)$ is fresh.
$s'; s''$	$F(s,l,C) = F(s', last(s'), F(s'', l, C))$, if $l \in labels(s'')$
	$F(s,l,C) = F(s', l, C)$, if $l \in labels(s')$
$l:$ if $(v)\ s'$ else s'	$F(s,l,C) = F(s', last(s'), C) \wedge F(s'', last(s''), C)$
	$F(s,l',C) = F(s', l', C)$, if $l' \in labels(s')$
	$F(s,l',C) = F(s'', l', C)$, if $l' \in labels(s'')$
$l:$ while $(v)\ \{s'\}$	$F(s,l,C) = C_{\text{fix}} \wedge C$, if $norm(F(s', last(s'), C_{\text{fix}} \wedge C)) = C_{\text{fix}}$
	$F(s,l',C) = F(s,l,C')$, if $F(s',l',C) = C'$ and $l' \in labels(s')$
$l:\ p = \text{malloc}$	$F(s,l,C) = C$
$l:\ x = *p$	$F(s,l,C) = C[l(id)/x]$, $l(id)$ is fresh.
$l:\ *p = x$	$F(s,l,C) = C(l_1(id_1),l,x)(l_2(id_2),l,x)\ldots(l_n(id_n),l,x)$ for all $l_1(id_1),\ldots,l_n(id_n)$ in C, where:

$$C(l_{\text{load}}(id), l, x) = \begin{cases} C & no\_alias(l, l_{\text{load}}) \\ C[x/l_{\text{load}}(id)] & \neg no\_alias(l, l_{\text{load}}) \wedge must\_alias(l, l_{\text{load}}) \\ C[x/l_{\text{load}}(id)] \wedge C & \neg no\_alias(l, l_{\text{load}}) \wedge \neg must\_alias(l, l_{\text{load}}) \end{cases}$$

Figure 6. Static analysis rules. The notation $C[e_a/e_b]$ denotes the symbolic condition obtained by replacing every occurrence of e_b in C with e_a. $norm(C)$ is the normalization function that transforms the condition C to an equivalent normalized condition.

sequences of statements propagates the symbolic expression set backwards through the statements in sequence.

Analysis of Load and Store Statements: The analysis of a load statement $x = *p$ replaces the assigned variable x with a materialized abstract value $l(id)$ that represents the loaded value. For input read statements, the analysis uses a newly materialized id to distinguish values read on different executions of the load statement.

The analysis of a store statement $*p = x$ uses the alias analysis to appropriately match the stored value x against all loads that may return that value. Specifically, the analysis locates all $l_i(id_i)$ atoms in C that either may or must load a value v that the store statement stores into the location p. If the alias analysis determines that the $l_i(id_i)$ expression must load x (i.e., the corresponding load statement will always access the last value that the store statement stored into location p), then the analysis of the store statement replaces all occurrences of $l_i(id_i)$ with x.

If the alias analysis determines that the $l_i(id_i)$ expression may load x (i.e., on some executions the corresponding load statement may load x, on others it may not), then the analysis produces two symbolic conditions: one with $l_i(id_i)$ replaced by x (for executions in which the load statement loads x) and one that leaves $l_i(id_i)$ in place (for executions in which the load statement loads a value other than x).

We note that, if the pointer analysis is imprecise, the symbolic condition may become intractably large. SIFT uses the DSA algorithm [17], a context-sensitive, unification-based pointer analysis. We found that, in practice, this analysis is precise enough to enable SIFT to efficiently analyze our benchmark applications (see Figure 14 in Section 5.2).

```
1  Input: Expression e
2  Output: Normalized expression e_norm
3
4  e_norm ←e
5  f_cnt ← {all →0}
6  l_cnt ← {all →0}
7  for a in Atoms(e) do
8    if a is in form f(id) then
9      nextid ←f_cnt(f) + 1
10     f_cnt ←f_cnt[f →nextid]
11     e_norm ←e_norm[*f(nextid)/f(id)]
12   else if a is in form l(id) then
13     nextid ←l_cnt(l) + 1
14     l_cnt ←l_cnt[l →nextid]
15     e_norm ←e_norm[*l(nextid)/l(id)]
16   end if
17 end
18 for a in Atoms(e_norm) do
19   if a is in form *f(id) then
20     e_norm ←e_norm[f(id)/ * f(id)]
21   else if a is in form *l(id) then
22     e_norm ←e_norm[l(id)/ * l(id)]
23   end if
24 end
```

Figure 8. Normalization function $norm(e)$. $Atom(e)$ iterates over the atoms in the expression e from left to right.

Analysis of Loop Statements: The analysis uses a fixed-point algorithm to synthesize the loop invariant C_{fix} required to analyze while loops. Specifically, the analysis of a state-

ment while$(x)\{s'\}$ computes a sequence of symbolic conditions C_i, where $C_0 = \emptyset$ and $C_i = norm(F(s', last(s'), C \wedge C_{i-1}))$. Conceptually, each successive symbolic condition C_i captures the effect of executing an additional loop iteration. The analysis terminates when it reaches a fixed point (i.e., when it has performed n iterations such that $C_n = C_{n-1}$). Here C_n is the discovered loop invariant. This fixed point correctly summarizes the effect of the loop (regardless of the number of iterations that it may perform).

The loop analysis normalizes the analysis result $F(s', last(s'), C \wedge C_{i-1})$ after each iteration. For a symbolic condition $C = \text{safe}(e_1) \wedge \ldots \wedge \text{safe}(e_n)$, the normalization of C is $norm(C) = remove\_dup(\text{safe}(norm(e_1)) \wedge \ldots \wedge \text{safe}(norm(e_n)))$, where $norm(e_i)$ is the normalization of each individual expression in C (using the algorithm presented in Figure 8) and $remove\_dup()$ removes duplicate conjuncts from the condition.

Normalization facilitates loop invariant discovery for loops that read input fields or load values via pointers. Each analysis of the loop body during the fixed point computation produces new materialized values $f(id)$ and $l(id)$ with fresh ids. The new materialized $f(id)$ represent input fields that the current loop iteration reads; the new materialized $l(id)$ represent values that the current loop iteration loads via pointers. The normalization algorithm appropriately renumbers these ids in the new symbolic condition so that the first appearance of each id is in lexicographic order. Because the normalization only renumbers ids, the normalized condition is equivalent to the original conditions (see Section 4.5). This normalization enables the analysis to recognize loop invariants that show up as equivalent successive analysis results that differ only in the materialized ids that they use to represent input fields and values accessed via pointers.

The above algorithm will reach a fixed point and terminate if it computes the symbolic condition of a value that depends on at most a statically fixed number of values from the loop iterations. For example, our algorithm is able to compute the symbolic condition of the size parameter value of the memory allocation in Figure 2 — the value of this size parameter depends only on the values of jpeg_width and jpeg_height, the current values of h_sample and v_sample, and the maximum values of h_sample and v_sample, each of which comes from one previous iteration of the loop at line 26-31.

Note that the algorithm will not reach a fixed point if it attempts to compute a symbolic condition that contains an unbounded number of values from different loop iterations. For example, the algorithm will not reach a fixed point if it attempts to compute a symbolic condition for the sum of a set of numbers computed within the loop (the sum depends on values from all loop iterations). To ensure termination, our current implemented algorithm terminates the analysis and fails to generate a symbolic condition C if it fails to reach a fixed point after ten iterations.

```
1  Input: A symbolic condition C
2  Output: F(l_call : v = call proc v_1 ... v_k, l_call, C),
3    where proc is defined as:
4    proc(a_1, a_2,...,a_k) { s; ret v_ret }
5  Where: l_1(id_1), l_2(id_2), ..., l_n(id_n)
6    are all atoms of the form l(id)
7    that appear in S.
8
9  R ← ∅
10 S_0 ←F(s, last(s), safe(v_ret))
11 for e_0 in exprs(S_0[v_1/a_1]...[v_n/a_n]) do
12   S_1 ←F(s, last(s), safe(l_1(id_1)))
13   for e_1 in exprs(S_1[v_1/a_1]...[v_n/a_n]) do
14     ...
15     S_n ←F(s_b, last(s), safe(l_n(id_n)))
16     for e_n in exprs(S_n[v_1/a_1]...[v_n/a_n]) do
17       e'_0 ←make_fresh(e_0, C)
18       ...
19       e'_n ←make_fresh(e_n, C)
20       R ←R ∧ C[e'_0/v]...[e'_i/label_i(id_i)]...
21     end
22     ...
23   end
24 end
25 F(l_call : v = call proc v_1 ... v_k, l_call, C) ← R
```

Figure 9. Procedure Call Analysis Algorithm. $make\_fresh(e, C)$ renumbers ids in e so that occurrences of $l(id)$ and $f(id)$ will not conflict with the condition C. $exprs(C)$ returns the set of expressions that appear in the conjuncts of C. For example, $expr(\text{safe}(e_1) \wedge \text{safe}(e_2)) = \{e_1, e_2\}$.

In practice, we expect that many programs may contain expressions whose values depend on an unbounded number of values from different loop iterations. Our analysis can successfully analyze such programs because it is demand driven — it only attempts to obtain precise symbolic representations of expressions that may contribute to the values of expressions in the analyzed symbolic condition C (which, in our current system, are ultimately derived from expressions that appear at memory allocation and block copy sites). Our experimental results indicate that our approach is, in practice, effective for this set of expressions, specifically because these expressions tend to depend on at most a fixed number of values from loop iterations.

3.3 Inter-procedural Analysis

Analyzing Procedure Calls: Figure 9 presents the inter-procedural analysis for procedure call sites. Given a symbolic condition C and a function call statement $l_{\text{call}} : v = call\, proc\ v_1 \ldots v_k$ that invokes a procedure $proc(a_1,$

a_2,...,a_k) { s; $ret\ v_{ret}$ }, the analysis computes $F(v = call\ proc\ v_1\ \ldots\ v_k, l_{\text{call}}, C)$.

Conceptually, the analysis performs two tasks. First, it replaces any occurrences of the procedure return value v in C (the symbolic condition after the procedure call) with symbolic expressions that represent the values that the procedure may return. Second, it transforms C to reflect the effect of any store instructions that the procedure may execute. Specifically, the analysis finds expressions $l(id)$ in C that represent values that 1) the procedure may store into a location p 2) that the computation following the procedure may access via a load instruction that may access (a potentially aliased version of) p. It then replaces occurrences of $l(id)$ in C with symbolic expressions that represent the corresponding values computed (and stored into p) within the procedure.

The analysis examines the invoked procedural body s to obtain the symbolic expressions that corresponds to the return value (see line 10) or the value of $l(id)$ (see lines 12 and 15). The analysis avoids redundant analysis of the invoked procedure by caching the analysis results $F(s, last(s), \text{safe}(v_{ret}))$ and $F(s, last(s), \text{safe}(l(id)))$ for reuse.

Note that symbolic expressions derived from an analysis of the invoked procedure may contain occurrences of the formal parameters $a_1, ..., a_k$. The interprocedural analysis translates these symbolic expressions into the name space of the caller by replacing occurrences of the formal parameters $a_1, ..., a_k$ with the corresponding actual parameters $v_1, ..., v_k$ from the call site (see lines 11, 13, and 16 in Figure 9).

Also note that the analysis carefully renumbers the ids in the symbolic expressions derived from an analysis of the invoked procedure before the replacements (see lines 17-19). This ensures that the occurances of $f(id)$ and $s(id)$ in the expressions are fresh in C.

Propagation to Program Entry: To derive the final symbolic condition at the start of the program, the analysis propagates the current symbolic condition up the call tree through procedure calls until it reaches the start of the program. When the propagation reaches the entry of the current procedure $proc$, the algorithm uses the procedure call graph to find all call sites that may invoke $proc$.

It then propagates the current symbolic condition C to the callers of $proc$, appropriately translating C into the naming context of the caller by substituting any formal parameters of $proc$ that appear in C with the corresponding actual parameters from the call site. The analysis continues this propagation until it has traced out all paths in the call graph from the initial critical site where the analysis started to the program entry point. The final symbolic condition C is the conjunction of the conditions derived along all of these paths.

3.4 Extension to C Programs

We next describe how to extend our analysis to real world C programs to generate input filters.

Identify Critical Sites: SIFT transforms the application source code into the LLVM intermediate representation (IR) [2], scans the IR to identify critical values (i.e., size parameters of memory allocation and block copy call sites) inside the developer specified module, and then performs the static analysis for each identified critical value. By default, SIFT recognizes calls to standard C memory allocation routines (such as malloc, calloc, and realloc) and block copy routines (such as memcpy). SIFT can also be configured to recognize additional memory allocation and block copy routines (for example, dMalloc in Dillo).

Bit Width and Signness: SIFT extends the analysis described above to track the bit width of each expression atom. It also tracks the sign of each expression atom and arithmetic operation and correctly handles extension and truncation operations (i.e., signed extension, unsigned extension, and truncation) that change the width of a bit vector. SIFT therefore faithfully implements the representation of integer values in the C program.

Function Pointers and Library Calls: SIFT uses its underlying pointer analysis [17] to disambiguate function pointers. It can analyze programs that invoke functions via function pointers.

The static analysis may encounter procedure calls (for example, calls to standard C library functions) for which the source code of the callee is not available. A standard way to handle this situation is to work with an annotated procedure declaration that gives the static analysis information that it can use to analyze calls to the procedure. If code for an invoked procedure is not available, by default SIFT currently synthesizes information that indicates that symbolic expressions are not available for the return value or for any values accessible (and therefore potentially stored) via procedure parameters (code following the procedure call may load such values). This information enables the analysis to determine if the return value or values accessible via the procedure parameters may affect the analyzed symbolic condition C. If so, SIFT does not generate a filter. Because SIFT is demand-driven, this mechanism enables SIFT to successfully analyze programs with library calls (all of our benchmark programs have such calls) as long as the calls do not affect the analyzed symbolic conditions.

Annotations for Input Read Statement: SIFT provides a declarative specification language that developers use to indicate which input statements read which input fields. In our current implementation these statements appear in the source code in comments directly below the C statement that reads the input field. See lines 10, 12, 15-16, and 18-

19 in Figure 2 for examples that illustrate the use of the specification language in the Swfdec example. The SIFT annotation generator scans the comments, finds the input specification statements, then inserts new nodes into the LLVM IR that contain the specified information. Formally, this information appears as procedure calls of the following form:

$$\mathrm{v} = \mathrm{SIFT\_Input}(\text{"field\_name"}, \mathrm{w});$$

where v is a program variable that holds the value of the input field with the field name field_name. The width (in bits) of the input field is w. The SIFT static analyzer recognizes such procedure calls as specifying the correspondence between input fields and program variables and applies the appropriate analysis rule for input read statements (see Figure 6).

Input Filter Generation: We prune any conjuncts that contain residual occurrences of abstract materialized values $l(id)$ in the final symbolic condition C. We also replace every residual occurance of program variables v with 0. Formally speaking, these residual occurances correspond to initial values of the program state $\overline{\sigma}$ and $\overline{h}$ in abstract semantics (see Section 4.3). The result condition C_{Inp} will contain only input value and constant atoms.

The filter operates as follows. It first uses an existing parser for the input format to parse the input and extract the input fields used in the input condition C_{Inp}. Open source parsers are available for a wide of input file formats, including all of the formats in our experimental evaluation [1]. These parsers provide a standard API that enables clients to access the parsed input fields.

The generated filter evaluates each conjunct expression in C_{Inp} by replacing each symbolic input variable in the expression with the corresponding concrete value from the parsed input. If an integer overflow may occur in the evaluation of any expression in C_{Inp}, the filter discards the input and optionally raises an alarm. For input field arrays such as h_sample and v_sample in the Swfdec example (see Section 2), the input filter enumerates all possible combinations of concrete values (see Figure 12 for the formal definition of condition evaluation). The filter discards the input if any combination can trigger the integer overflow error.

Given multiple symbolic conditions generated from multiple critical program points, SIFT can create a single efficient filter that first parses the input, then checks the input against all symbolic conditions in series on the parsed input. This approach amortizes the overhead of reading the input (in practice, reading the input consumes essentially all of the time required to execute the filter, see Figure 15) over all of the symbolic condition checks.

4. Soundness of the Static Analysis

We next formalize our static analysis algorithm on the core language in Figure 4 and discuss the soundness of the analysis. We focus on the intraprocedural analysis and omit a discussion of the interprocedural analysis as it uses standard techniques based on summary tables.

4.1 Dynamic Semantics of the Core Language

Program State: We define the program state $(\sigma, \rho, \varsigma, \varrho, Inp)$ as follows:

σ: Var $\to$ (Loc + Int + {undef}) $\quad$ ς: Var $\to$ Bool
ρ: Loc $\to$ (Loc + Int + {undef}) $\quad$ ϱ: Loc $\to$ Bool
Inp: InputField $\to \mathcal{P}$(Int)

σ and ρ map variables and memory locations to their corresponding values. We use undef to represent unintialized values. We define that if any operand of an arithmetic operation is undef, the result of the operation is also undef. Inp represents the input file, which is unchanged during the execution. ς maps each variable to a boolean flag, which tracks whether the computation that generates the value of the variable (including all sub-computations) generates an overflow. ϱ maps each memory location to a boolean overflow flag similar to ς.

The initial states σ_0 and ρ_0 map all variables and locations to undef. The initial states of ς_0 and ϱ_0 map all variables and locations to false. The values of uninitialized variables and memory locations are undefined as per the C language specification standard.

Small Step Rules: Figure 10 presents the small step dynamic semantics of the language. Note that in Figure 10, $overflow(a, b, op)$ is a function that returns true if and only if the computation $a\, op\, b$ causes overflow. A main point of departure from standard languages is that we also update ς and ϱ to track overflow errors during each execution step. For example, the op rule in Figure 10 appropriately updates the overflow flag of x in ς by checking whether the computation that generates the value of x (including the sub-computations that generates the value of y and z) results in an overflow condition.

4.2 Soundness of the Pointer Analysis

Our analysis uses an underlying pointer analysis [17] to analyze programs that use pointers. We formally state our assumptions about the soundness of the underlying pointer alias analysis as follows:

Definition 1 (Soundness of no_alias and $must\_alias$)**.** *Given a sequence of execution*

$$\langle s_0, \sigma_0, \rho_0, \varsigma_0, \varrho_0 \rangle \longrightarrow \langle s_1, \sigma_1, \rho_1, \varsigma_1, \varrho_1 \rangle \longrightarrow \cdots$$

and two labels l_{store}, l_{load}, where l_{store} is the label for the store statement s_{store} such that s_{store} = "$l_{store} : *p = x$"

$$\text{read}\ \frac{c \in Inp(f) \quad \sigma' = \sigma[x \to c] \quad \varsigma' = \varsigma[x \to \text{false}]}{\langle l : x = read(f), \sigma, \rho, \varsigma, \varrho, Inp\rangle \longrightarrow \langle \text{nil: skip}, \sigma', \rho, \varsigma', \varrho, Inp\rangle}$$

$$\text{const}\ \frac{\sigma' = \sigma[x \to c] \quad \varsigma' = \varsigma[x \to \text{false}]}{\langle l : x = c, \sigma, \rho, \varsigma, \varrho, Inp\rangle \longrightarrow \langle \text{nil: skip}, \sigma', \rho, \varsigma', \varrho, Inp\rangle}$$

$$\text{assign}\ \frac{\sigma' = \sigma[x \to \sigma(y)] \quad \varsigma' = \varsigma[x \to \varsigma(y)]}{\langle l : x = y, \sigma, \rho, \varsigma, \varrho, Inp\rangle \longrightarrow \langle \text{nil: skip}, \sigma', \rho, \varsigma', \varrho, Inp\rangle}$$

$$\text{malloc}\ \frac{\xi \in Loc \quad \xi \text{ is fresh} \quad \sigma' = \sigma[p \to \xi] \quad \varsigma' = \varsigma[p \to \text{false}]}{\langle l : p = \text{malloc}, \sigma, \rho, \varsigma, \varrho, Inp\rangle \longrightarrow \langle \text{nil: skip}, \sigma', \rho, \varsigma', \varrho, Inp\rangle}$$

$$\text{seq-1}\ \frac{}{\langle \text{null: skip}; s, \sigma, \rho, \varsigma, \varrho, Inp\rangle \longrightarrow \langle s, \sigma, \rho, \varsigma, \varrho, Inp\rangle}$$

$$\text{load}\ \frac{\sigma(p) = \xi \quad \xi \in Loc \quad \sigma' = \sigma[x \to \rho(\xi)] \quad \varsigma' = \varsigma[x \to \varrho(\xi)]}{\langle l : x = *p, \sigma, \rho, \varsigma, \varrho, Inp\rangle \longrightarrow \langle \text{nil: skip}, \sigma', \rho, \varsigma', \varrho, Inp\rangle}$$

$$\text{seq-2}\ \frac{\langle s, \sigma, \rho, \varsigma, \varrho, Inp\rangle \longrightarrow \langle s'', \sigma', \rho', \varsigma', \varrho', Inp\rangle}{\langle s; s', \sigma, \rho, \varsigma, \varrho, Inp\rangle \longrightarrow \langle s''; s', \sigma', \rho', \varsigma', \varrho', Inp\rangle}$$

$$\text{store}\ \frac{\sigma(p) = \xi \quad \xi \in Loc \quad \rho' = \rho[\xi \to \sigma(x)] \quad \varrho' = \varrho[\xi \to \varsigma(x)]}{\langle l : *p = x, \sigma, \rho, \varsigma, \varrho, Inp\rangle \longrightarrow \langle \text{nil: skip}, \sigma, \rho', \varsigma, \varrho', Inp\rangle}$$

$$\text{op}\ \frac{\sigma(y) \notin Loc \quad \sigma(z) \notin Loc \quad b = \varsigma(y) \vee \varsigma(z) \vee overflow(\sigma(y), \sigma(z), op)}{\langle l : x = y\ op\ z, \sigma, \rho, \varsigma, \varrho, Inp\rangle \longrightarrow \langle \text{nil: skip}, \sigma[x \to \sigma(y)\ op\ \sigma(z)], \rho, \varsigma[x \to b], \varrho, Inp\rangle}$$

$$\text{if-t}\ \frac{\sigma(x) \neq 0}{\langle l : \text{if}\ (x)\ s\ \text{else}\ s', \sigma, \rho, \varsigma, \varrho, Inp\rangle \longrightarrow \langle s, \sigma, \rho, \varsigma, \varrho, Inp\rangle}$$

$$\text{if-f}\ \frac{\sigma(x) = 0}{\langle l : \text{if}\ (x)\ s\ \text{else}\ s', \sigma, \rho, \varsigma, \varrho, Inp\rangle \longrightarrow \langle s', \sigma, \rho, \varsigma, \varrho, Inp\rangle}$$

$$\text{while-f}\ \frac{\sigma(x) = 0}{\langle l : \text{while}\ (x)\ \{s\}, \sigma, \rho, \varsigma, \varrho, Inp\rangle \longrightarrow \langle \text{nil: skip}, \sigma, \rho, \varsigma, \varrho, Inp\rangle}$$

$$\text{while-t}\ \frac{\sigma(x) \neq 0 \quad s' = s; l : \text{while}\ (x)\ \{s\}}{\langle l : \text{while}\ (x)\ \{s\}, \sigma, \rho, \varsigma, \varrho, Inp\rangle \longrightarrow \langle s', \sigma, \rho, \varsigma, \varrho, Inp\rangle}$$

Figure 10. The small step operational semantics of the language. "nil" is a special label reserved by the semantics.

and l_{load} is the label for the load statement s_{load} such that s_{load} = "$l_{load} : x' = *p'$", we have:

$$no\_alias(l_{store}, l_{load}) \rightarrow \forall_{i,first(s_i)=l_{store}} \forall_{j,first(s_j)=l_{load},i<j}\ \sigma_i(p) \neq \sigma_j(p')$$

$$must\_alias(l_{store}, l_{load}) \rightarrow \forall_{i,first(s_i)=l_{store}} \forall_{j,first(s_j)=l_{load},i<j} ((\forall_{k,i<k<j} first(s_k) \neq l_{store}) \rightarrow (\sigma_i(p) = \sigma_j(p')))$$

Intuitively, 1) $no\_alias(l_{\text{store}}, l_{\text{load}})$ = true only if the load statement at label l_{load} will never retrieve a value stored by the store statement at label l_{store}; 2) $must\_alias(l_{\text{store}}, l_{\text{load}})$ = true only if the load statement at label l_{load} will always retrieve a value from the last memory location that the store statement at label l_{store} manipulates

4.3 Abstract Semantics

We next define an abstract semantics that allows us to certify the operation of our static analysis algorithm. The key difference between the abstract and original semantics is that the abstract semantics enables the non-deterministic branch selection of conditional statements (i.e., if- and while-statements) and the use of a non-deterministic memory model. This accurately captures how our analysis ignores control flow conditions and uses an underlying pointer analysis to handle pointers.

Abstract Program State: We define abstract program state $(\overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp)$ as follows:

$\overline{\sigma}$: Var $\to$ Int $\overline{\varsigma}$: Var $\to$ Bool
$\overline{\hbar}$: LoadLabel $\to \mathcal{P}$(Int $\times$ Bool)

Intuitively, $\overline{\sigma}$ and $\overline{\varsigma}$ are the counterparts of σ and ς in the original semantics, but $\overline{\sigma}$ and $\overline{\varsigma}$ only track values and flags for variables that have integer values. $\overline{\hbar}$ maps the label of each load statement to the set of values that the load statement may obtain from the memory.

We define the initial state $\overline{\sigma}_0$ and $\overline{\varsigma}_0$ to map all variables to 0 and false respectively. We define the initial state $\overline{\hbar}_0$ to map all labels of load statements to the empty set.

Small Step Rules: Figure 11 presents the small step rules for the abstract semantics.

The rules for conditional, loop, malloc, load, and store statements capture the main difference between the abstract semantics and the original semantics. The rules for conditional and while statements (if-t, if-f, while-t, and while-f rules) in abstract semantics ignore branch conditions and allow non-deterministic execution of either path. The rule for store statements maintains the state $\overline{\hbar}$ according to the alias information. The rule for load statements non-deterministically returns an element from the corresponding set in $\overline{\hbar}$.

4.4 Relationship of the Original and the Abstract Semantics

We next formally state the relationship of the original semantics and the abstract semantics as the follow theorem. We present the proof sketch of this theorem in the appendix.

Theorem 2. *Given any execution trace in the original semantics*

$$\langle s_0, \sigma_0, \rho_0, \varsigma_0, \varrho_0\rangle \longrightarrow \langle s_1, \sigma_1, \rho_1, \varsigma_1, \varrho_1\rangle \longrightarrow \cdots,$$

there is an execution trace in the abstract semantics

$$\langle s_0, \overline{\sigma}_0, \overline{\varsigma}_0, \overline{\hbar}_0\rangle \longrightarrow_a \langle s_1, \overline{\sigma}_1, \overline{\varsigma}_1, \overline{\hbar}_1\rangle \longrightarrow_a \ldots$$

$$\text{read}\ \frac{c \in Inp(f) \quad \overline{\sigma}' = \overline{\sigma}[x \to c] \quad \overline{\varsigma}' = \overline{\varsigma}[x \to \text{false}]}{\langle l : x = read(f), \overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp\rangle \longrightarrow_a \langle \text{nil:skip}, \overline{\sigma}', \overline{\varsigma}', \overline{\hbar}, Inp\rangle}$$

$$\text{constant}\ \frac{\overline{\sigma}' = \overline{\sigma}[x \to c] \quad \overline{\varsigma}' = \overline{\varsigma}[x \to \text{false}]}{\langle l : x = c, \overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp\rangle \longrightarrow_a \langle \text{nil: skip}, \overline{\sigma}', \overline{\varsigma}', \overline{\hbar}, Inp\rangle}$$

$$\text{assign}\ \frac{}{\langle l : x = y, \overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp\rangle \longrightarrow_a \langle \text{nil: skip}, \overline{\sigma}[x \to \overline{\sigma}(y)], \overline{\varsigma}[x \to \overline{\varsigma}(y)], \overline{\hbar}, Inp\rangle}$$

$$\text{if-t}\ \frac{}{\langle l : \text{if}\ (x)\ s\ \text{else}\ s', \overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp\rangle \longrightarrow_a \langle s, \overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp\rangle}$$

$$\text{op}\ \frac{b = \overline{\varsigma}(y) \vee \overline{\varsigma}(z) \vee overflow(\overline{\sigma}(y), \overline{\sigma}(z), op) \quad \overline{\sigma}' = \overline{\sigma}[x \to \overline{\sigma}(y)\ op\ \overline{\sigma}(z)]}{\langle l : x = y\ op\ z, \overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp\rangle \longrightarrow_a \langle \text{nil: skip}, \overline{\sigma}', \overline{\varsigma}[x \to b], \overline{\hbar}, Inp\rangle}$$

$$\text{if-f}\ \frac{}{\langle l : \text{if}\ (x)\ s\ \text{else}\ s', \overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp\rangle \longrightarrow_a \langle s', \overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp\rangle}$$

$$\text{while-f}\ \frac{}{\langle l : \text{while}\ (x)\ \{s\}, \overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp\rangle \longrightarrow_a \langle \text{nil: skip}, \overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp\rangle}$$

$$\text{while-t}\ \frac{}{\langle l : \text{while}\ (x)\ \{s\}, \overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp\rangle \longrightarrow_a \langle s; \text{while}\ (x)\ \{s\}, \overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp\rangle}$$

$$\text{seq-1}\ \frac{}{\langle \text{nil: skip}; s, \overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp\rangle \longrightarrow_a \langle s, \overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp\rangle}$$

$$\text{seq-2}\ \frac{\langle s, \overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp\rangle \longrightarrow_a \langle s'', \overline{\sigma}', \overline{\varsigma}', \overline{\hbar}', Inp\rangle}{\langle s; s', \overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp\rangle \longrightarrow_a \langle s''; s', \overline{\sigma}', \overline{\varsigma}, \overline{\hbar}', Inp\rangle}$$

$$\text{malloc}\ \frac{}{\langle l : p = malloc, \overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp\rangle \longrightarrow_a \langle \text{nil: skip}, \overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp\rangle}$$

$$\text{load}\ \frac{(c, b) \in \overline{\hbar}(l) \quad \overline{\sigma}' = \overline{\sigma}[x \to c] \quad \overline{\varsigma}' = \overline{\varsigma}[x \to b]}{\langle l : x = *p, \overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp\rangle \longrightarrow_a \langle \text{nil: skip}, \overline{\sigma}', \overline{\varsigma}', \overline{\hbar}, Inp\rangle}$$

$$\text{store}\ \frac{\overline{\hbar}' \text{ satisfies (*)}}{\langle l : *p = x, \overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp\rangle \longrightarrow_a \langle \text{nil: skip}, \overline{\sigma}, \overline{\varsigma}, \overline{\hbar}', Inp\rangle}$$

$$\forall l_{\text{load}} \in \text{LoadLabel} : \overline{\hbar}'(l_{\text{load}}) = \begin{cases} \overline{\hbar}(l_{\text{load}}) & no\_alias(l, l_{\text{load}}) \\ \{(\overline{\sigma}(x), \overline{\varsigma}(x))\} & \neg no\_alias(l, l_{\text{load}}) \wedge must\_alias(l, l_{\text{load}}) \\ \{(\overline{\sigma}(x), \overline{\varsigma}(x))\} \cup \overline{\hbar}(l_{\text{load}}) & \neg no\_alias(l, l_{\text{load}}) \wedge \neg must\_alias(l, l_{\text{load}}) \end{cases} \quad (*)$$

Figure 11. The small step abstract semantics. "nil" is a special label reserved by the semantics.

such that the following conditions hold

$$\forall_i \forall_{x \in Var} \\ (\sigma_i(x) \in \mathit{Int} \to (\sigma_i(x) = \overline{\sigma}_i(x) \wedge \varsigma_i(x) = \overline{\varsigma}_i(x))) \quad (1)$$

$$\forall_{i, first(s_i)=l, l \in LoadLabel, left(s_i)=\text{“}l:x=*p\text{”}} \\ (\rho_i(\sigma_i(p)) \in \mathit{Int} \to (\rho_i(\sigma_i(p)), \varrho_i(\sigma_i(p))) \in \overline{\hbar}_i(l)) \quad (2)$$

where $left(s)$ denotes an utility function that returns the leftmost statement in s if s is a sequential composite and returns s if s is not a sequential composite.

The intuition of Condition 1 is that σ_i and $\overline{\sigma}_i$ as well as ς_i and $\overline{\varsigma}_i$ always agree on the variables holding integer values. The intuition of Condition 2 is that $\overline{\hbar}_i(l)$ corresponds to the possible values that the corresonding load statement of the label l may obtain from the memory. Thus when a load statement is executed, the obtained integer value should be in the corresponding set in $\overline{\hbar}_i$.

4.5 Soundness of the Analysis

Evaluation of the Condition C: Our static analysis maintains and propagates the symbolic condition C. Figure 12 defines the evaluation rules of the symbolic condition C over the abstract program states. We use $(\overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp) \models C$ to denote that the abstract program state $(\overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp)$ satisfies the condition C.

Note that the evaluation rule for safe$(e_1\ op\ e_2)$ not only ensures that the overflow error does not occur at the last computation step, but also ensures no overflow error occurs in sub-computations recursively.

Also note that, intuitively, atoms in the form of $f(id)$ and $l(id)$ correspond to a set of possible values from an input field or a load statement ($Inp(f)$ or $\overline{\hbar}(l)$). Thus the evaluation rules enumerate all possible value bindings and make sure that each value binding satisfies the condition.

The definition of the evaluation rules also indicates the normalization algorithm in Section 3.2 that renumbers ids in a symbolic condition C does not change the semantic meaning of the condition. Therefore for a given symbolic condition C, the normalization algorithm produces an equivalent condition $norm(C)$.

Soundness of the Analysis over the Abstract Semantics: We formally state the soundness of our analysis over the abstract semantics as follows.

Theorem 3. *Given a series of statements s_i, a program point $l \in labels(s_i)$ and a start condition C, our analysis generates a condition $F(s_i, l, C)$ such that if $(\overline{\sigma}_i, \overline{\varsigma}_i, \overline{\hbar}_i, Inp) \models F(s_i, l, C)$, then $\langle s_i, \overline{\sigma}_i, \overline{\varsigma}_i, \overline{\hbar}_i\rangle \longrightarrow_a^* \langle s_{j-1}, \overline{\sigma}_{j-1}, \overline{\varsigma}_{j-1}, \overline{\hbar}_{j-1}\rangle \longrightarrow_a \langle s_j, \overline{\sigma}_j, \overline{\varsigma}_j, \overline{\hbar}_j\rangle \wedge first(s_{j-1}) = l$ implies $(\overline{\sigma}_j, \overline{\varsigma}_j, \overline{\hbar}_j, Inp) \models C$.*

This guarantees that if the abstract program state before executing s_i satisfies $F(s_i, l, C)$, then the abstract program state at the program point after the statement of the label l will always satisfy C (" $\longrightarrow_a^*$ " represents to execute the program for an arbitrary number of steps in the abstract semantics).

Soundness of the Analysis over the Original Semantics: Because of the consistency of the abstract semantics and the original semantics (see Section 4.3), we can derive the following soundness property of our analysis over the original

$$\frac{\forall_{c \in Inp(f)} : (\overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp) \models C[c/f(id)]}{(\overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp) \models C} \qquad \frac{\forall_{(c,b) \in \overline{\hbar}(l)} : (\overline{\sigma}[tmp \rightarrow c], \overline{\varsigma}[tmp \rightarrow b], \overline{\hbar}, Inp) \models C[tmp/l(id)] \quad tmp \text{ is fresh in } C}{(\overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp) \models C}$$

$$\frac{(\overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp) \models C \quad (\overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp) \models safe(e)}{(\overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp) \models C \wedge safe(e)} \qquad \frac{(\overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp) \models safe(e_1) \wedge safe(e_2) \quad overflow([[e_1]](\overline{\sigma}), [[e_2]](\overline{\sigma}), op) = \text{false}}{(\overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp) \models safe(e_1 \; op \; e_2)}$$

$$\frac{\overline{\varsigma}(x) = false}{(\overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp) \models safe(x)} \qquad [[c]](\overline{\sigma}) = c \qquad [[x]](\overline{\sigma}) = \overline{\sigma}(x) \qquad [[e_1 \; op \; e_2]](\overline{\sigma}) = [[e_1]](\overline{\sigma}) \; op \; [[e_2]](\overline{\sigma})$$

Figure 12. Condition evaluation rules.

semantics based on the soundness property over the abstract semantics:

Theorem 4. *Given a program s_0, a program point $l \in labels(s_0)$, and a program variable v, our analysis generates a condition $C = F(s_0, l, safe(v))$ such that if $(\overline{\sigma}_0, \overline{\varsigma}_0, \overline{\hbar}_0, Inp) \models C$, then $\langle s_0, \sigma_0, \rho_0, \varsigma_0, \varrho_0 \rangle \longrightarrow^* \langle s_{n-1}, \sigma_{n-1}, \rho_{n-1}, \varsigma_{n-1}, \varrho_{n-1} \rangle \longrightarrow \langle s_n, \sigma_n, \rho_n, \varsigma_n, \varrho_n \rangle \wedge first(s_{n-1}) = l \wedge \sigma_n(v) \in$ Int implies $\varsigma_n(v) =$ false.*

This guarantees that if the input satisfies the generated condition C (note that $\overline{\sigma}_0$, $\overline{\varsigma}_0$, and $\overline{\hbar}_0$ are predefined constant initial state in Section 4.3), then for any execution in the original semantics (" $\longrightarrow^*$ " represents to execute the program for an arbitrary number of steps in the original semantics), at the program point after the statement of the label l, as long as the variable v holds an integer value (not an undefined value due to unintialized access), the computation history for obtaining this integer value contains no overflow error.

5. Experimental Results

We evaluate SIFT on modules from five open source applications: VLC 0.8.6h (a network media player), Dillo 2.1 (a lightweight web browser), Swfdec 0.5.5 (a flash video player), Swftools 0.9.1 (SWF manipulation and generation utilities), and GIMP 2.8.0 (an image manipulation application). Each application uses a publicly available input format specification and contains at least one known integer overflow vulnerability (described in either the CVE database or the Buzzfuzz paper [13]). All experiments were conducted on an Intel Xeon X5363 3.00GHz machine running Ubuntu 12.04.

5.1 Methodology

Input Format and Module Selection: For each application, we used SIFT to generate filters for the input format that triggers the known integer overflow vulnerability. We therefore ran SIFT on the module that processes inputs in that format. The generated filters nullify not only the known vulnerabilities, but also any integer overflow vulnerabilities at any of the 52 memory allocation or block copy sites in the modules for which SIFT was able to generate symbolic conditions (recall that there are 58 critical sites in these modules in total).

Application	Distinct Fields	Relevant Fields
VLC	25	2
Dillo	47	3
Swfdec	219*	6
png2swf	47	4
jpeg2swf	300	2
GIMP	189	2

Figure 13. The number of distinct input fields and the number of relevant input fields for analyzed input formats. (*) For Swfdec the second column shows the number of distinct fields in embedded JPEG images in collected SWF files.

Input Statement Annotation: After selecting each module, we added annotations to identify the input statements that read relevant input fields (i.e., input fields that may affect the values of critical expressions at memory allocation or block copy sites). Figure 13 presents, for each module, the total number of distinct fields in our collected inputs for each format, the number of annotated input statements (in all of the modules the number of relevant fields equals the number of annotated input statements — each relevant field is read by a single input statement). We note that the number of relevant fields is significantly smaller than the total number of distinct fields (reflecting the fact that typically only a relatively small number of fields in each input format may affect the sizes of allocated or copied memory blocks).

The maximum amount of time required to annotate any module was approximately half an hour (Swfdec). The total annotation time required to annotate all benchmarks, including Swfdec, was less than an hour. This annotation effort reflects the fact that, in each input format, there are only a relatively small number of relevant input fields.

Filter Generation and Test: We next used SIFT to generate a single composite input filter for each analyzed module. We then downloaded at least 6000 real-world inputs for each input format, and ran all of the downloaded inputs through

the generated filters. There were no false positives (the filters accepted all of the inputs).

Vulnerability and Filter Confirmation: For each known integer overflow vulnerability, we collected a test input that triggered the integer overflow. We confirmed that each generated composite filter, as expected, discarded the input because it correctly recognized that the input would cause an integer overflow.

5.2 Analysis and Filter Evaluation

Figure 14 presents static analysis and filter generation results. This figure contains a row for each analyzed module. The first column (Application) presents the application name, the second column (Module) identifies the analyzed module within the application. The third column (# of IR) presents the number of analyzed statements in the LLVM intermediate representation. This number of statements includes not only statements directly present in the module, but also statements from analyzed code in other modules invoked by the original module.

The fourth column (Total) presents the total number of memory allocation and block copy sites in the analyzed module. The fifth column (Input Relevant) presents the number of memory allocation and block copy sites in which the size of the allocated or copied block depends on the values of input fields. For these modules, the sizes at 49 of the 58 sites depend on the values of input fields. The sizes at the remaining nine sites are unconditionally safe — SIFT verifies that they depend only on constants embedded in the program (and that there is no overflow when the sizes are computed from these constants).

The sixth column (Inside Loop) presents the number of memory allocation and block copy sites in which the size parameter depends on variables that occurred inside loops. For these modules, the sizes at 29 of the 58 sites depend on loops relevant variables, for which SIFT needs to compute loop invariants to generate input filters.

The seventh column (Max Condition Size) presents, for each application module, the maximum number of conjunts in any symbolic condition that occurs in the analysis of that module. The conditions are reasonably compact (and more than compact enough to enable an efficient analysis) — the maximum condition size over all modules is less than 500.

The final column (Analysis Time) presents the time required to analyze the module and generate a single composite filter for all of the successfully analyzed critical sites. The analysis times for all modules are less than a second.

SIFT is unable to generate symbolic conditions for 6 of the 58 call sites. For two of these sites (one in Swfdec and one in png2swf), the two expressions contain subexpressions whose value depends on an unbounded number of values from loop iterations. To analyze such expressions, our analysis currently requires an upper bound on the number of loop iterations. Such an upper bound could be provided, for example, by additional analysis or developer annotations. The remaining four expressions (two in png2swf and two in jpeg2swf) depend on the return value from strlen(). SIFT is not currently designed to analyze such expressions.

Application	Format	# of Input	Average Time
VLC	WAV	10976	3ms (3ms)
Dillo	PNG	18983	16ms (16ms)
Swfdec	SWF	7240	6ms (5ms)
png2swf	PNG	18983	16ms (16ms)
jpeg2swf	JPEG	6049	4ms (4ms)
GIMP	GIF	19647	9ms (9ms)

Figure 15. Generated Filter Results.

```
1  #define __EVEN( x ) (((x)%2 != 0 ) ? ((x)+1) : (x))
2
3  static int Open( vlc_object_t * p_this ) {
4    ...
5    // search format chunk and read its size
6    if( ChunkFind( p_demux, "fmt ", &i_size ) )
7    {
8      msg_Err( p_demux, "cannot find 'fmt ' chunk" );
9      goto error;
10   }
11   /* i_size = SIFT_input("fmt_size", 32); */
12   ...
13   // where integer overflow happens.
14   p_wf_ext = malloc( __EVEN( i_size ) + 2 );
15 }
```

Figure 17. The simplified source code from VLC with annotations inside comments.

For each input format, we used a custom web crawler to locate and download at least 6000 inputs in that format. The web crawler starts from a Google search page for the file extension of the specific input format, then follows links in each search result page to download files in the correct format.

Figure 15 presents, for each generated filter, the number of downloaded input files and the average time required to filter each input. We present the average times in the form Xms (Yms), where Xms is the average time required to filter an input and Yms is the average time required to read in the input (but not apply the integer overflow check). These data show that essentially all of the filter time is spent reading in the input.

5.3 Vulnerability Case Studies

In Section 2 we showed how SIFT handles the integer overflow vulnerability in Swfdec. We next investigate how SIFT handles the remaining five known vulnerabilities in our benchmark applications. Figure 16 presents the symbolic conditions that SIFT generates for each of the five vulnerabilities in the analyzed modules.

5.3.1 VLC

Application	Module	# of IR	Total	Input Relevant	Inside Loop	Max Condition Size	Analysis Time
VLC	demux/wav.c	1.5k	5	3	0	2	<0.1s
Dillo	png.c	39.1k	4	3	3	410	0.8s
Swfdec	jpeg/*.c	8.4k	22	19	2	144	0.2s
png2swf	all	11.0k	21	18	18	16	0.2s
jpeg2swf	all	2.5k	4	4	4	2	<0.1s
GIMP	file-gif-load.c	3.2k	2	2	2	2	<0.1s

Figure 14. Static Analysis and Filter Generation Results

VLC	$safe((fmt\_size^{[32]} + 1^{[32]}) + 2^{[32]}) \wedge safe(fmt\_size^{[32]} + 2^{[32]})$
png2swf	$\wedge_{c=1}^{4} safe((c^{[32]} \times png\_width^{[32]}) \times png\_height^{[32]} + 65536^{[32]})$
jpeg2swf	$safe((jpeg\_width^{[32]} \times jpeg\_height^{[32]}) \times 4^{[32]})$
Dillo	$\wedge_{c=1}^{4}(safe(((png\_width^{[32]} \times (c^{[32]} \times sext(png\_bitdepth^{[8]}, 32)) + 7^{[32]}) >> 3^{[32]}) \times png\_height_h t^{[32]}) \wedge$ $safe(png\_width^{[32]} \times ((c^{[32]} \times sext(png\_bitdepth^{[8]}, 32)) >> 3^{[32]}) \times png\_height^{[32]}))$
GIMP	$safe((gif\_width^{[32]} \times gif\_height^{[32]}) \times 2^{[32]}) \wedge safe(gif\_width^{[32]} \times gif\_height^{[32]} \times 4^{[32]})$

Figure 16. The symbolic condition C in the bit vector form for VLC, Swftools-png2swf, Swftools-jpeg2swf, Dillo and GIMP. The superscript indicates the bit width of each expression atom. "$sext(v, w)$" is the signed extension operation that transforms the value v to the bit width w.

The VLC `wav.c` module contains an integer overflow vulnerability (CVE-2008-2430) when parsing WAV sound inputs. Figure 17 presents (a simplified version of) the source code that is related to this error. When VLC parses the format chunk of a WAV input, it first reads the input field fmt_size, which indicates the size of the format chunk (line 6). VLC then allocates a buffer to hold the format chunk (line 14 in Figure 17). A large fmt_size field value (for example, 0xfffffffe) will cause an overflow to occur when VLC computes the buffer size.

We annotate the source code to specify where the module reads the fmt_size input field (line 11). SIFT then analyzes the module to obtain the symbolic condition C (Figure 16), which soundly summarizes how VLC computes the buffer size from input fields.

5.3.2 Dillo

Dillo contains an integer overflow vulnerability (CVE-2009-2294) in its png module. Figure 18 presents the simplified source code for this example. Dillo uses the libpng library to read PNG images. The libpng runtime calls png_process_data() (line 2) to process each PNG image. This function then calls png_push_read_chunk() (line 11) to process each chunk in the PNG image. When the libpng runtime reads the first data chunk (the IDAT chunk), it calls the Dillo callback png_datainfo_callback() (lines 66-75) in the Dillo PNG processing module. There is an integer overflow vulnerability at line 73 where Dillo calculates the size of the image buffer as png->rowbytes*png->height. On a 32-bit machine, inputs with large width and height fields can cause the image buffer size calculation to overflow. In this case Dillo allocates an image buffer that is smaller than required and eventually writes beyond the end of the allocated buffer.

Figure 16 presents the symbolic condition C for Dillo. C soundly takes intermediate computations over all execution paths into consideration, including the switch branch at lines 45-59 that sets the variable png_ptr->channels and PNG_ROWBYTES macro at lines 26-29. Note that the constant $c^{[32]}$ in C corresponds to the possible values of png_ptr->channels, which are between 1 and 4.

5.3.3 Swftools

Swftools is a set of utilities for creating and manipulating SWF files. Swftools contains two tools png2swf and jpeg2swf, which transform PNG and JPEG images to SWF files. Each of these two tools contains an integer overflow vulnerability (CVE-2010-1516).

Figure 19 presents (a simplified version of) the source code that contains the png2swf vulnerability. When processing PNG images, Swftools calls getPNG() (lines 20-43) at png2swf.c:763 to read the PNG image into memory. getPNG() first calls png_read_header() (lines 1-18) to locate and read the header chunk which contains the PNG metadata. It then uses the metadata information to calculate the length of the image data at png.h:502 (lines 39-40). There is no bounds check on the width and the height value from the header chunk before this calculation. On a 32-bit machine, a PNG image with large width and height values will trigger the integer overflow error.

We annotate lines 7 and 10 the statements that read input fields png_width and png_height and use SIFT to de-

```c
// libpng main data process function.
void png_process_data(png_structp png_ptr,
 png_infop info_ptr, ...) {
  ...
  while (png_ptr->buffer_size) {
    // This is a wrapper for png_push_read_chunk
    png_process_some_data(png_ptr, info_ptr);
  }
}
// chunk handler dispatcher
void png_push_read_chunk(png_structp png_ptr,
 png_infop info_ptr) {
  if (!png_memcmp(png_ptr->chunk_name,png_IHDR,4)){
    ...
    png_handle_IHDR(png_ptr, info_ptr, ...);
  }
  ...
  else if (!png_memcmp(png_ptr->chunk_name,
   png_IDAT, 4)) {
    ...
    // Datainfo callback is called
    png_push_have_info(png_ptr, info_ptr);
    ...
  }
}
#define PNG_ROWBYTES(pixel_bits,width)\
  ((pixel_bits)>=8?\
  ((width)*(((png_uint_32)(pixel_bits))>>3)):\
  ((((width)*((png_uint_32)(pixel_bits)))+7)>>3))
void png_handle_IHDR(png_structp png_ptr,
 png_infop info_ptr, ...) {
  ...
  // read individual png fields from input buffer
  width = png_get_uint_31(png_ptr, buf);
  /* width = SIFT_input("png_width", 32); */
  height = png_get_uint_31(png_ptr, buf + 4);
  /* height = SIFT_input("png_height", 32); */
  bit_depth = buf[8];
  /* bit_depth = SIFT_input("png_bitdepth", 8); */
  ...
  png_ptr->width = width;
  png_ptr->height = height;
  png_ptr->bit_depth = (png_byte)bit_depth;
  ...
  switch (png_ptr->color_type) {
     case PNG_COLOR_TYPE_GRAY:
     case PNG_COLOR_TYPE_PALETTE:
        png_ptr->channels = 1;
        break;
     case PNG_COLOR_TYPE_RGB:
        png_ptr->channels = 3;
        break;
     case PNG_COLOR_TYPE_GRAY_ALPHA:
        png_ptr->channels = 2;
        break;
     case PNG_COLOR_TYPE_RGB_ALPHA:
        png_ptr->channels = 4;
        break;
  }
  png_ptr->pixel_depth = (png_byte)(
   png_ptr->bit_depth * png_ptr->channels);
  png_ptr->rowbytes = PNG_ROWBYTES(
   png_ptr->pixel_depth, png_ptr->width);
}
// Dillo datainfo initialization callback
static void Png_datainfo_callback(png_structp png_ptr,
 ...) {
  DilloPng *png;
  png = png_get_progressive_ptr(png_ptr);
  ...
  // where the overflow happens
  png->image_data = (uchar_t *) dMalloc(
    png->rowbytes * png->height);
  ...
}
```

Figure 18. The simplified source code from Dillo and libpng with annotations inside comments.

```c
static int png_read_header(FILE*fi,
 struct png_header*header) {
  ...
  while(png_read_chunk(&id, &len, &data, fi)) {
    if(!strncmp(id, "IHDR", 4)) {
      ...
      header->width = data[0]<<24|data[1]<<16|
       data[2]<<8|data[3];
      /*header->width=SIFT_input("png_width",32);*/
      header->height = data[4]<<24|data[5]<<16|
       data[6]<<8|data[7];
      /*header->height=SIFT_input("png_height",32);*/
      ...
    }
    ...
  }
  ...
}

EXPORT int getPNG(const char*sname, int*destwidth,
 int*destheight, unsigned char**destdata) {
  ...
  unsigned long int imagedatalen;
  ...
  if(!png_read_header(fi, &header)) {
    fclose(fi);
    return 0;
  }
  ...
  if(header.mode==3 || header.mode==0) bypp = 1;
    else if(header.mode == 4) bypp = 2;
    else if(header.mode == 2) bypp = 3;
    else if(header.mode == 6) bypp = 4;
    else {
      ...
      return 0;
    }

  imagedatalen = bypp * header.width *
   header.height + 65536;
  imagedata = (unsigned char*)malloc(imagedatalen);
  ...
}
```

Figure 19. The simplified source code from png2swf in swftools with annotations inside comments.

rive the symbolic condition for this vulnerability. Figure 16 presents the symbolic condition C.

jpeg2swf contains a similar integer overflow vulnerability when processing JPEG images. At jpeg2swf.c:171 jpeg2swf first calls the libjpeg API to read jpeg image. At jpeg2swf.c:173, jpeg2swf then immediately calculates the size of a memory buffer for holding the jpeg file in its own data structure. Because it directly uses the input width and height values in the calculation without range checks, large width and height values may cause overflow errors. Figure 16 presents the symbolic expression condition C for jpeg2swf.

5.3.4 GIMP

GIMP contains an integer overflow vulnerability (CVE-2012-3481) in its GIF loading plugin file-gif-load.c. When GIMP opens a GIF file, it calls load_image at file-gif-load.c:335 to load the entire GIF file into memory. For each individual image in the GIF file, this function first reads the

image metadata information, then calls $\mathrm{ReadImage}$ to process the image. At $\mathrm{file\text{-}gif\text{-}load.c{:}1064}$, the plugin calculates the size of the image output buffer as a function of the product of the width and height values from the input. Because it uses these values directly without range checks, large height and width fields may cause an integer overflow. In this case GIMP may allocate a buffer smaller than the required size.

We annotate the source code based on the GIF specification and use SIFT to derive the symbolic condition for this vulnerability. Figure 16 presents the generated symbolic expression condition C.

5.4 Discussion

The experimental results highlight the combination of properties that, together, enable SIFT to effectively nullifying potential integer overflow errors at memory allocation and block copy sites. SIFT is efficient enough to deploy in production on real-world modules (the combined program analysis and filter generation times are always under a second), the analysis is precise enough to successfully generate input filters for the majority of memory allocation and block copy sites, the results provide encouraging evidence that the generated filters are precise enough to have few or even no false positives in practice, and the filters execute efficiently enough to deploy with acceptable filtering overhead.

6. Related Work

Weakest Precondition: Madhavan et. al. present an approximate weakest precondition analysis to verify the absence of null dereference errors in Java programs [21]. The underlying analysis domain tracks whether or not variables may contain null references. To obtain acceptable precision for the null dereference verification problem, the technique incorporates null-dereference checks from conditional statements into the propagated conditions.

Because SIFT focuses on integer overflow errors, the underlying analysis domain (symbolic arithmetic expressions) and propagation rules are significantly more complex. SIFT also does not incorporate checks from conditional statements, a design decision that, for the integer overflow problem, produces efficient and accurate filters. Also the problems are different — SIFT generates filters to eliminate security vulnerabilities, while Madhavan et. al. focus on verifying the absence of null dereferences.

Flanagan et. al. presents a general intraprocedural weakest precondition analysis for generating verification conditions for ESC/JAVA programs [12]. SIFT differs in that it focuses on integer overflow errors. Because of this focus, SIFT can synthesize its own loop invariants (Flanagan et. al. rely on developer-provided invariants). In addition, SIFT is interprocedural, and uses the analysis results to generate sound filters that nullify integer overflow errors.

Anomaly Detection: Anomaly detection techniques generate (unsound) input filters by empirically learning properties of successfully or unsuccessfully processed inputs [14, 16, 19, 23, 25, 26, 30, 31]. Web-based anomaly detection [16, 26] uses input features (e.g. request length and character distributions) from attack-free HTTP traffic to model normal behaviors. HTTP requests that contain features that violate the model are flagged as anomalous and dropped. Similarly, Valeur et al [30] propose a learning-based approach for detecting SQL-injection attacks. Wang et al [31] propose a technique that detects network-based intrusions by examining the character distribution in payloads. Perdisci et al [25] propose a clustering-based anomaly detection technique that learns features from malicious traces (as opposed to benign traces). Input rectification learns properties of inputs that the application processes successfully, then modifies inputs to ensure that they satisfy the learned properties [20].

Two key differences between SIFT and these techniques are that SIFT statically analyzes the application, not its inputs, and takes all execution paths into account to generate a sound filter.

Static Analysis for Finding Integer Errors: Several static analysis tools have been proposed to find integer errors [6, 27, 32]. KINT [32], for example, analyzes individual procedures, with the developer optionally providing procedure specifications that characterize the value ranges of the parameters. KINT also unsoundly avoids the loop invariant synthesis problem by replacing each loop with the loop body (in effect, unrolling the loop once). Despite substantial effort, KINT reports a large number of false positives [32].

SIFT addresses a different problem: it is designed to nullify, not detect, overflow errors. In pursuit of this goal, it uses an interprocedural analysis, synthesizes symbolic loop invariants, and soundly analyzes all execution paths to produce a sound filter.

Symbolic Test Generation: DART [15] and KLEE [5] use symbolic execution to automatically generate test cases that can expose errors in an application. IntScope [29] and SmartFuzz [22] are symbolic execution systems specifically for finding integer errors. It would be possible to combine these systems with previous input-driven filter generation techniques to obtain filters that discard inputs that take the discovered path to the error. As discussed previously, SIFT differs in that it considers all possible paths so that its generated filters come with a soundness guarantee that if an input passes the filter, it will not exploit the integer overflow error.

Runtime Check and Library Support: To alleviate the problem of false positives, several research projects have focused on runtime detection tools that dynamically insert runtime checks before integer operations [3, 7, 11, 34]. Another similar technique is to use safe integer libraries such as

SafeInt [18] and CERT's IntegerLib [28] to perform sanity checks at runtime. Using these libraries requires that developers rewrite existing code to use safe versions of integer operations.

However, the inserted code typically imposes non-negligible overhead. When integer errors happen in the middle of the program execution, these techniques usually raise warnings and terminate the execution, which effectively turn integer overflow attacks into DoS attacks. SIFT, in contrast, inserts no code into the application and blocks inputs that exploit integer overflow vulnerabilities to avoid the attacks completely.

Benign Integer Overflows: In some cases, developers may intentionally write code that contains benign integer overflows [29, 32]. A potential concern is that techniques that nullify overflows may interfere with the intended behavior of such programs [29, 32]. Because SIFT focuses on critical memory allocation and block copy sites that are unlikely to have such intentional integer overflows, SIFT is unlikely to nullify benign integer overflows and therefore unlikely interfere with the intended behavior of the program.

7. Conclusion

Integer overflow errors can lead to security vulnerabilities. SIFT analyzes how the application computes integer values that appear at memory allocation and block copy sites to generate input filters that discard inputs that may trigger overflow errors in these computations. Our results show that SIFT can quickly generate efficient and precise input filters for the vast majority of memory allocation and block copy call sites in our analyzed benchmark modules.

References

[1] Hachoir. `http://bitbucket.org/haypo/hachoir/wiki/Home`.

[2] The LLVM compiler infrastructure. `http://www.llvm.org/`.

[3] D. Brumley, T. Chiueh, R. Johnson, H. Lin, and D. Song. Rich: Automatically protecting against integer-based vulnerabilities. *Department of Electrical and Computing Engineering*, page 28, 2007.

[4] D. Brumley, H. Wang, S. Jha, and D. Song. Creating vulnerability signatures using weakest preconditions. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, CSF '07', pages 311–325, Washington, DC, USA, 2007. IEEE Computer Society.

[5] C. Cadar, D. Dunbar, and D. Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[6] E. Ceesay, J. Zhou, M. Gertz, K. Levitt, and M. Bishop. Using type qualifiers to analyze untrusted integers and detecting security flaws in c programs. *Detection of Intrusions and Malware & Vulnerability Assessment*, pages 1–16, 2006.

[7] R. Chinchani, A. Iyer, B. Jayaraman, and S. Upadhyaya. Archerr: Runtime environment driven program safety. *Computer Security–ESORICS 2004*, pages 385–406, 2004.

[8] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: securing software by blocking bad input. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07. ACM, 2007.

[9] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of internet worms. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05. ACM, 2005.

[10] W. Cui, M. Peinado, and H. J. Wang. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *Proceedings of 2007 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2007.

[11] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in c/c++. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 760–770. IEEE Press, 2012.

[12] C. Flanagan and J. B. Saxe. Avoiding exponential explosion: generating compact verification conditions. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '01', pages 193–205, New York, NY, USA, 2001. ACM.

[13] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009.

[14] D. Gao, M. K. Reiter, and D. Song. On gray-box program tracking for anomaly detection. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04. USENIX Association, 2004.

[15] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[16] C. Kruegel and G. Vigna. Anomaly detection of web-based attacks. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03. ACM, 2003.

[17] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 278–289, New York, NY, USA, 2007. ACM.

[18] D. LeBlanc. Integer handling with the c++ safeint class. *urlhttp://msdn. microsoft. com/en-us/library/ms972705*, 2004.

[19] F. Long, V. Ganesh, M. Carbin, S. Sidiroglou, and M. Rinard. Automatic input rectification. ICSE '12, 2012.

[20] F. Long, V. Ganesh, M. Carbin, S. Sidiroglou, and M. Rinard. Automatic input rectification. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 80–90, Piscataway, NJ, USA, 2012. IEEE Press.

[21] R. Madhavan and R. Komondoor. Null dereference verification via over-approximated weakest pre-conditions analysis. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 1033–1052, New York, NY, USA, 2011. ACM.

[22] D. Molnar, X. C. Li, and D. A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. Usenix Security'09.

[23] D. Mutz, F. Valeur, C. Kruegel, and G. Vigna. Anomalous system call detection. *ACM Transactions on Information and System Security*, 9, 2006.

[24] J. Newsome, D. Brumley, and D. X. Song. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *NDSS*, 2006.

[25] R. Perdisci, W. Lee, and N. Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10. USENIX Association, 2010.

[26] W. Robertson, G. Vigna, C. Kruegel, and R. A. Kemmerer. Using generalization and characterization techniques in the anomaly-based detection of web attacks. In *Proceedings of the 13 th Symposium on Network and Distributed System Security (NDSS)*, 2006.

[27] D. Sarkar, M. Jagannathan, J. Thiagarajan, and R. Venkatapathy. Flow-insensitive static analysis for detecting integer anomalies in programs. In *IASTED*, 2007.

[28] R. Seacord. *The CERT C secure coding standard*. Addison-Wesley Professional, 2008.

[29] W. Tielei, W. Tao, L. Zhiqiang, and Z. Wei. IntScope: Automatically Detecting Integer Overflow Vulnerability In X86 Binary Using Symbolic Execution. In *16th Annual Network & Distributed System Security Symposium*, 2009.

[30] F. Valeur, D. Mutz, and G. Vigna. A learning-based approach to the detection of sql attacks. In *DIMVA 2005*, 2005.

[31] K. Wang and S. J. Stolfo. Anomalous payload-based network intrusion detection. In *RAID*, 2004.

[32] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. Kaashoek. Improving integer security for systems with kint. In *OSDI*. USENIX Association, 2012.

[33] X. Wang, Z. Li, J. Xu, M. K. Reiter, C. Kil, and J. Y. Choi. Packet vaccine: black-box exploit detection and signature generation. CCS '06. ACM, 2006.

[34] C. Zhang, T. Wang, T. Wei, Y. Chen, and W. Zou. Intpatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. *Computer Security–ESORICS 2010*, pages 71–86, 2010.

A. Proof Sketch of the Relationship between the Original Semantics and the Abstract Semantics

A.1 The Alias Analyses and the Abstract Semantics

In order to prove the above relationship between the original semantics and the abstract semantics, we introduce the following lemma that states the property between alias relationships and the abstract semantics.

Lemma 5. *Given an execution trace* $\langle s_0, \overline{\sigma}_0, \overline{\varsigma}_0, \overline{\hbar}_0\rangle \longrightarrow_a \langle s_1, \overline{\sigma}_1, \overline{\varsigma}_1, \overline{\hbar}_1\rangle \longrightarrow_a \cdots$ *in the abstract semantics, we have*

$$\begin{array}{l}\forall_{i,first(s_i)=l,left(s_i)=\text{“}l:*p=x\text{”}}\forall_{j,i<j}\forall_{l_{load}\in LoadLabel}\\ \quad(\neg no\_alias(l, l_{load})\\ \quad\wedge(\forall_{i<k<j,first(s_k)\in StoreLabel} : \neg must\_alias(first(s_k), l_{load}))\\ \quad\rightarrow((\overline{\sigma}_i(x), \overline{\varsigma}_i(x)) \in \overline{\hbar}_j(l_{load}))).\end{array} \tag{3}$$

The intuition of Condition 3 is that $\overline{\sigma}_i(x)$ and $\overline{\varsigma}_i(x)$ are the integer value and the overflow flag of the variable x that the corresponding store statement of the label l accesses. If the store statement and the corresponding load statement of the label l_{load} may access the same memory location and all later store statements in the execution trace until the state $\langle s_j, \overline{\sigma}_j, \overline{\varsigma}_j, \overline{\hbar}_j\rangle$ do not have $must\_alias$ relationship with the load statement, then the pair of the integer value and the overflow flag $(\overline{\sigma}_i(x), \overline{\varsigma}_i(x))$ should be in $\overline{\hbar}_j(l_{\text{load}})$.

Proof. We can prove Condition 3 with the induction on the number of steps of the execution n.

When $n = 0$, the condition trivially holds. Now consider the induction case where $n > 0$. If $first(s_{n-1}) \notin StoreLabel$, then based on the small step rules of semantics, $\overline{\hbar}_{n-1} = \overline{\hbar}_n$. It is straintforward to apply the induction rule to prove the condition.

If $first(s_{n-1}) \in StoreLabel$ and $s_{n-1} = \text{“}l : *p' = x'\text{”}$, based on the induction, what we need to prove is the case where $j = n$:

$$\begin{array}{l}\forall_{i,first(s_i)=l,left(s_i)=\text{“}l:*p=x\text{”},i<n}\\ \forall_{l_{\text{load}}\in LoadLabel}(\neg no\_alias(l, l_{\text{load}})\wedge\\ (\forall_{i<k<n,first(s_k)\in StoreLabel} : \neg must\_alias(first(s_k), l_{\text{load}}))\\ \rightarrow((\overline{\sigma}_i(x), \overline{\varsigma}_i(x)) \in \overline{\hbar}_n(l_{\text{load}})))\end{array}$$

If $i = n - 1$, from the small step rule of the abstract semantics of the store statement, we can prove that $\forall_{l_{\text{load}}\in LoadLabel}(\neg no\_alias(l, l_{\text{load}})) \rightarrow ((\overline{\sigma}_i(x), \overline{\varsigma}_i(x)) \in \overline{\hbar}_n(l_{\text{load}}))$. Therefore the conditon holds.

If $i < n - 1$, from the small step rule of the store statement we can prove that $\forall_{l_{\text{load}}\in LoadStatement}(\neg must\_alias(first(s_{n-1}), l_{\text{load}})) \rightarrow$

$(\overline{\hbar}_{n-1}(l_{\text{load}}) \subseteq \overline{\hbar}_n(l_{\text{load}}))$. From the induction rule, we can show that $(\overline{\sigma}_i(x), \overline{\varsigma}_i(x)) \in \overline{\hbar}_{n-1}(l_{\text{load}})$. Therefore, we can prove the condition holds. □

A.2 The Relationship between the Original Semantics and the Abstract Semantics

Theorem 2. *Given any execution trace in the original semantics*

$$\langle s_0, \sigma_0, \rho_0, \varsigma_0, \varrho_0\rangle \longrightarrow \langle s_1, \sigma_1, \rho_1, \varsigma_1, \varrho_1\rangle \longrightarrow \cdots,$$

there is an execution trace in the abstract semantics

$$\langle s_0, \overline{\sigma}_0, \overline{\varsigma}_0, \overline{\hbar}_0\rangle \longrightarrow_a \langle s_1, \overline{\sigma}_1, \overline{\varsigma}_1, \overline{\hbar}_1\rangle \longrightarrow_a \cdots$$

such that the following conditions hold

$$\forall_i \forall_{x \in \mathit{Var}} \quad (\sigma_i(x) \in \mathit{Int} \rightarrow (\sigma_i(x) = \overline{\sigma}_i(x) \wedge \varsigma_i(x) = \overline{\varsigma}_i(x))) \tag{1}$$

$$\forall_{i, first(s_i)=l, l\in LoadLabel, left(s_i)=\text{“}l:x=*p\text{”}} \quad (\rho_i(\sigma_i(p)) \in \mathit{Int} \rightarrow (\rho_i(\sigma_i(p)), \varrho_i(\sigma_i(p))) \in \overline{\hbar}_i(l)) \tag{2}$$

where $left(s)$ denotes an utility function that returns the leftmost statement in s if s is a sequential composite and returns s if s is not a sequential composite.

Proof. The proof of Conditions 1, 2 can be done with induction on the number of steps in the execution trace n.

For the base case $n = 0$, we simply verify that initial states satisfy Conditions 1 2.

For the induction case $n > 0$, we already have $\overline{\sigma}_0, \ldots, \overline{\sigma}_{n-1}$, $\overline{\varsigma}_0, \ldots, \overline{\varsigma}_{n-1}$, $\overline{\hbar}_0, \ldots, \overline{\hbar}_{n-1}$ that satisfy the conditions by the induction rule. We first construct $\overline{\sigma}_n$, $\overline{\varsigma}_n$, and $\overline{\hbar}_n$ using the corresponding small step rule in abstract semantics, and then prove the construction satisfies the conditions.

Condition 1: If $first(s_{n-1}) \notin LoadLabel$, the proof is straightforward. For example, if $first(s_{n-1}) = l$, $left(s_{n-1}) =$ “$l : x = y\, op\, z$”, based on the small step rule of the original semantics we know that:

$$\begin{aligned}\sigma_n &= \sigma_{n-1}[x \rightarrow \sigma_{n-1}(y)\, op\, \sigma_{n-1}(z)]\\ \varsigma_n &= \varsigma_{n-1}[x \rightarrow \varsigma_{n-1}(y) \vee \varsigma_{n-1}(z) \vee \\ &\qquad overflow(\sigma_{n-1}(y), \sigma_{n-1}(z), op)]\end{aligned}$$

and we can construct using the corresponding rule in the abstract semantics:

$$\begin{aligned}\overline{\sigma}_n &= \overline{\sigma}_{n-1}[x \rightarrow \overline{\sigma}_{n-1}(y)\, op\, \overline{\sigma}_{n-1}(z)]\\ \overline{\varsigma}_n &= \overline{\varsigma}_{n-1}[x \rightarrow \overline{\varsigma}_{n-1}(y) \vee \overline{\varsigma}_{n-1}(z) \vee \\ &\qquad overflow(\overline{\sigma}_{n-1}(y), \overline{\sigma}_{n-1}(z), op)]\end{aligned}$$

By the induction rule we have $\forall_{v\in \text{Var}}\, \sigma_{n-1}(v) \in \text{Int} \rightarrow \sigma_{n-1}(v) = \overline{\sigma}_{n-1}(v)$, so it is easy to show that:

$$\forall_{v\in Var, v\neq x} \sigma_n(x) \in Int \rightarrow \sigma_n(x) = \overline{\sigma}_n(x)$$

Also consider

$$\begin{aligned}&\sigma_n(x) \in \text{Int} \rightarrow\\ &(\sigma_{n-1}(y) \in \text{Int} \wedge \sigma_{n-1}(z) \in \text{Int}) \rightarrow\\ &(\sigma_{n-1}(y) = \overline{\sigma}_{n-1}(y) \wedge \sigma_{n-1}(z) = \overline{\sigma}_{n-1}(z)) \rightarrow\\ &((\sigma_{n-1}(y)\, op\, \sigma_{n-1}(z)) = (\overline{\sigma}_{n-1}(y)\, op\, \overline{\sigma}_{n-1}(z))) \rightarrow\\ &(\sigma_n(x) = \overline{\sigma}_n(x))\end{aligned}$$

We can do the proof similarly for ς and $\overline{\varsigma}$. Therefore Condition 1 holds.

If $first(s_{n-1}) \in LoadLabel$ and $left(s_{n-1}) =$ “$l : x = *p$”, based on the semantic rules of the load statement, we know that:

$$\begin{aligned}\sigma_n &= \sigma_{n-1}[x \rightarrow \rho_{n-1}(\sigma_{n-1}(p))]\\ \varsigma_n &= \varsigma_{n-1}[x \rightarrow \varrho_{n-1}(\sigma_{n-1}(p))]\end{aligned}$$

From the induction rule of Condition 2, we know that: $\rho_{n-1}(\sigma_{n-1}(p)) \in \text{Int} \rightarrow (\rho_{n-1}(\sigma_{n-1}(p)), \varrho_{n-1}(\sigma_{n-1}(p))) \in \overline{\hbar}(l)$. Therefore, if $\rho_{n-1}(\sigma_{n-1}(p)) \in \text{Int}$, it is possible to construct $\overline{\sigma}_n$ and $\overline{\varsigma}_m$ as follows based on the abstract semantic rule of the load statement:

$$\begin{aligned}\overline{\sigma}_n &= \overline{\sigma}_{n-1}[x \rightarrow \rho_{n-1}(\sigma_{n-1}(p))]\\ \overline{\varsigma}_n &= \overline{\varsigma}_{n-1}[x \rightarrow \varrho_{n-1}(\sigma_{n-1}(p))]\end{aligned}$$

From the induction rule of Condition 1, we know that $\sigma_{n-1} = \overline{\sigma}_{n-1}$ and $\varsigma_{n-1} = \overline{\varsigma}_{n-1}$. These facts together are enough to show that Condition 1 holds.

Condition 2: If $first(s_n) \notin LoadLabel$, the proof is trivial by using induction rule. Next we are going to sketch the proof of Condition 2, when $first(s_n) \in LoadLabel$ and $left(s_n) =$ “$l : x = *p$”.

We try to find a program state $\langle s_m, \sigma_m, \varsigma_m, \rho_m, \varrho_m\rangle$ in prior execution steps, such that $m < n$, $first(s_m) = l'$, $left(s_m) =$ “$l' : *p' = x'$”, $\sigma_m(p') = \sigma_n(p)$, and

$$\forall_{m<k<n, left(s_k)=\text{“}*p''=x''\text{”}} : \sigma_k(p'') \neq \sigma_n(p)$$

Intuitively, the $left(s_m)$ is the latest store statement that acesses the memory location p. Therefore, the load statement $left(s_n)$ should obtain the value that $left(s_m)$ stores. We can use a simple induction on the original small step semantics to prove that: 1) if we cannot find such intermediate state, we can prove that $\rho_n(\sigma_n(p)) \notin \text{Int}$; 2) or $\sigma_m(x') = \rho_n(\sigma_n(p))$ and $\varsigma_m(x') = \varrho_n(\sigma_n(p))$.

Then with the soundness property definition of alias predicate $must\_alias$ and no_alias, we prove that $\neg no\_alias(l', l)$ and $\forall_{m<k<n, first(s_k)\in StoreLabel} \neg must\_alias(first(s_k), l)$. Therefore, as a direct implication of Condition 3, we know that $(\overline{\sigma}_m(x'), \overline{\varsigma}_m(x')) \in \overline{\hbar}_n(s_n)$. With Condition 1 we have proved, we get $(\sigma_m(x') \in \text{Int}) \rightarrow ((\sigma_m(x'), \varsigma_m(x')) \in \overline{\hbar}_n(s_n))$. Therefore, we have $\rho_n(\sigma_n(p)) \in \text{Int} \rightarrow (\rho_n(\sigma_n(p)), \varrho_n(\sigma_n(p))) \in \overline{\hbar}_n(s_n)$. Condition 2 holds. □