Sound Input Filter Generation for Integer Overflow Errors

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

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

Our results show that SIFT successfully analyzes (and therefore generates sound input filters for) 56 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.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program Analysis; D.4.6 [*Operating Systems*]: Security and Protection

General Terms Security; Program Analysis

Keywords Integer Overflow; Abstraction; Soundness

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 [25, 33, 35].

1.1 Previous Filter Generation Systems

Standard filter generation systems start with an input that triggers an error [10–12, 26, 36]. 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 the best of our knowledge, the only previous technique that uses this approach [5] 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 *execution path constraints*, i.e., 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.

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]

POPL '14, January 22–24, 2014, San Diego, CA, USA.

ACM 978-1-4503-2544-8/14/01.
http://dx.doi.org/10.1145/2535838.2535888

[1] As is standard in the field, SIFT is designed to work with programs that do not access uninitialized 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 memory when the program computes a value of `e`, then no integer overflow occurs during the evaluation

1.3 No Execution Path Constraints

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. This design decision has the following consequences:

- **Sound and Efficient Analysis:** Ignoring execution path constraints improves the efficiency of the analysis because it eliminates the need to track the different constraints that may appear on the many different execution paths to each memory allocation or block copy site. Indeed, this efficiency is critical to the soundness of SIFT's analysis — in general, there may be an intractably large or even statically unbounded number of paths to a given memory allocation or block copy site. Attempting to enumerate all of these different execution paths to derive the complete set of execution path constraints is clearly infeasible and a major source of the unsoundness of previous techniques.
- **Efficient Filters:** Because SIFT ignores checks from conditional statements, it 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:** One potential concern is that the program may contain safety checks that enable it to safely process inputs that would otherwise trigger overflows. Ignoring these safety checks may cause the generated filter to discard inputs even though the program can process them safely.

  Our experimental results show that, in practice, 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.

1.4 Input Fields With Multiple Instantiations

Input files often contain multiple instantiations of the same input field (for example, when the input file contains repeated components that have the same format). SIFT works with an abstraction in which free variables in the propagated symbolic expressions (and the final symbolic condition) represent *all* instantiations of the corresponding input fields that they reference — i.e., SIFT does not attempt to determine the precise correspondence between variables and different instantiations of the same input field.

This design decision simplifies and extends the range of the analysis. All variables that reference the same input field are interchangeable (because they all represent all possible instantiations of the corresponding input field). SIFT can therefore successfully analyze programs for which it is not possible to statically determine the precise correspondence between variables and different instantiations of the same input field. This design decision also enables SIFT to analyze programs in which a single variable may reference different instantiations of the same input field over the course of the execution. Interchangeability also enables a new expression normalization algorithm that renumbers variables during the analysis of loops to obtain loop invariant expressions (see Section 3.2). These expressions soundly characterize the effect (on the propagated symbolic expression) of loops that may access multiple potentially different instantiations of the same input field.

One final consequence of this design decision is that the generated input filter must check all combinations of input field instantiations when it checks for potential overflows (see Section 3.5). Our experimental results show that, for our benchmark applications, this approach causes, at most, negligible overhead — the generated filters spend essentially all of their time reading the input.

of `e` (including the evaluations of intermediate expressions that contribute to the final value of the critical expression).

1.5 Pointer Analysis and Precondition Generation

SIFT groups pointers into equivalence sets based on an analysis of the potential aliasing relationships between different pointers. To analyze a load statement that loads a value via a pointer, SIFT generates a new variable that represents the loaded value. In the propagated symbolic conditions, each such variable represents *all* values that may be stored via *any* alias of the pointer.

This decision produces an appropriately imprecise abstraction that enables SIFT to successfully analyze programs for which it is not possible to statically determine the precise value that each variable references. It also enables SIFT to work with any sound pointer or alias analysis that provides SIFT with the aliasing information required to soundly update the propagated symbolic condition at statements that access values via pointers (see Section 3.2). To the best of our knowledge, this is the first paper to show how to soundly incorporate an arbitrary off-the-shelf alias or pointer analysis into a precondition generation algorithm.

1.6 SIFT Usage Model

SIFT implements 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 automatically 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.

1.7 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 56 of these 58 critical sites (SIFT's static analysis was unable to derive symbolic conditions for the remaining two critical sites, see Section 5.2 for more details). These applications contain six known integer overflow vulnerabilities at their critical sites. SIFT's filters nullify all of these vulnerabilities.

Analysis and Filter Generation Times. 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 at 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.

1.8 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.

  Unlike all previous techniques of which we are aware, SIFT is sound — because it takes all execution paths into consideration, if an input passes the generated filter, it will not trigger an integer overflow error at any analyzed site. Also unlike previous techniques, SIFT generates efficient filters — because SIFT ignores execution path constraints, the generated filters perform tens of operations (as opposed to previous techniques, which incorporate execution path constraints and therefore perform tens of thousands or more operations).
- **Sound and Efficient 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. Unlike standard precondition generation techniques, the SIFT static analysis does not incorporate checks from the program's conditional statements — it instead works only with the arithmetic operations that contribute directly to the values of the critical expressions.
- **Input Fields With Multiple Instantiations:** We present a novel abstraction for input fields with multiple instantiations. This abstraction enables SIFT to analyze programs for which it is impossible to statically determine the precise correspondence between variables and different instantiations of the same input field. With this abstraction, all variables that reference the same input field are interchangable (because they all represent all instantiations of that input field). This interchangability enables a new expression normalization technique that SIFT deploys to automatically obtain invariants for loops that access the values of input fields.
- **Pointer Analysis and Precondition Generation:** We also present a novel abstraction for values that load statements access via pointers. This abstraction enables SIFT to analyze programs for which it is impossible to statically determine the precise value that each pointer references. We believe that this paper is the first to show how to soundly incorporate an arbitrary off-the-shelf alias or pointer analysis into a precondition generation algorithm.
- **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 rest of this paper is organized as follows. Section 2 presents a motivating 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 discusses 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 1 presents (simplified) source code from Swfdec. When Swfdec opens an SWF file with embedded JPEG images, it calls `jpeg_decoder_decode()` (line 1 in Figure 1) to decode each JPEG image in the file. This function in turn calls the function `jpeg_decoder_start_of_frame()` (line 7) to read the image metadata and the function `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 multiple instantiations of the `h_sample` field and the `v_sample` field. Swfdec computes the

$$
\begin{aligned}
C: \quad & \mathrm{safe}((((\mathrm{sext}(jpeg\_width^{[16]}, 32) + 8^{[32]} \times \mathrm{sext}(h\_sample\langle 1\rangle^{[4]}, 32) - 1^{[32]})/(8^{[32]} \times \mathrm{sext}(h\_sample\langle 1\rangle^{[4]}, 32)) \\
& \quad \times 8^{[32]} \times \mathrm{sext}(h\_sample\langle 1\rangle^{[4]}, 32))/(\mathrm{sext}(h\_sample\langle 1\rangle^{[4]}, 32)/\,\mathrm{sext}(h\_sample\langle 2\rangle^{[4]}, 32))) \\
& \quad \times (((\mathrm{sext}(jpeg\_height^{[16]}, 32) + 8^{[32]} \times \mathrm{sext}(v\_sample\langle 1\rangle^{[4]}, 32) - 1^{[32]})/(8^{[32]} \times \mathrm{sext}(v\_sample\langle 1\rangle^{[4]}, 32)) \\
& \quad \times 8^{[32]} \times \mathrm{sext}(v\_sample\langle 1\rangle^{[4]}, 32))/(\mathrm{sext}(v\_sample\langle 1\rangle^{[4]}, 32)/\,\mathrm{sext}(v\_sample\langle 2\rangle^{[4]}, 32))))
\end{aligned}
$$

Figure 2. 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. "$\mathrm{sext}(v, w)$" is the signed extension operation that transforms the value v to the bit width w.

```
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 1. Simplified Swfdec source code. Input statement annotations appear in comments.

maximum values of these instantiations in the loop at lines 26–31. It then uses these maximum 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 computation of the expression for the size of the buffer uses the `max_h_sample` variable and the `max_v_sample` variable, which correspond to the maximum values of all instantiations of the `h_sample` field and the `v_sample` field. It is impossible to statically determine the precise instantiation that these two variables represent. To overcome this challenge, SIFT uses a novel abstraction in which variables in the propagated symbolic condition represent all instantiations of the corresponding input fields.

Second, the source code snippet contains many load/store statements that access values derived from input fields via pointers. To reason soundly about these load/store statements, SIFT uses an abstraction that enables SIFT to incorporate an off-the-shelf alias analysis [20].

Finally, Swfdec reads the input fields (lines 14 and 17) and computes the size of the allocated memory block (lines 45-46) in the loops at different procedures. SIFT therefore uses an interprocedural analysis that propagates the symbolic conditions across procedure boundaries to obtain precise symbolic conditions. SIFT also deploys a combination of a fixed point analysis and a novel expression normalization technique to obtain loop invariants that successfully characterize the effect of loops on propagated expressions.

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

Source Code Annotations: SIFT provides a 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 1. 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 17 to the corresponding input field values. Note that the input fields `h_sample` and `v_sample` may contain multiple instantiations, which 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 2. We use the notation "$\mathrm{safe}(e)$" in Figure 2 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 2, the superscripts indicate the bit width of each expression atom. $\mathrm{sext}(v, w)$ 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 2 omits this information.

Note that SIFT soundly handles the loops that access the instantiations of the input fields `h_sample` and `v_sample`. In the resulting final symbolic condition C, $h\_sample\langle 1\rangle$ represents the instantiation of the input field `h_sample` that corresponds to the program variable `max_h_sample`, while $h\_sample\langle 2\rangle$ represents the instantiation that corresponds to the program variable `dec->components[i].h_sample`. SIFT does not attempt to determine the precise instantiations that $h\_sample\langle 1\rangle$ and $h\_sample\langle 2\rangle$ represent. Instead, SIFT conservatively assumes that

$h\_sample\langle 1\rangle$ and $h\_sample\langle 2\rangle$ may independently represent any instantiation of the field `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 of the form $\text{safe}(e)$. In general, however, different program paths may compute different values for the critical expression. In this case, the final symbolic condition C will contain multiple conjuncts of the form $\text{safe}(e)$. Each conjuct captures one of the ways that the program computes the value of the critical expression.

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 $\text{safe}(e)$ in C, the input triggers integer overflow errors when evaluating e (including all subexpressions). The generated filter extracts all instantiations 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 instantiations of the input fields `h_sample` and `v_sample` to consider all possible bindings of $h\_sample\langle 1\rangle$, $h\_sample\langle 2\rangle$, $v\_sample\langle 1\rangle$, and $v\_sample\langle 2\rangle$ 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 SIFT static analysis algorithm. We have implemented our static analysis for C programs using the LLVM Compiler Infrastructure [3].

3.1 Core Language and Notation

$$
\begin{array}{rcl}
s & := & \ell\colon\ x = \texttt{read(}f\texttt{)} \mid \ell\colon\ x = c \mid \ell\colon\ x = y \mid \\
& & \ell\colon\ x = y\ op\ z \mid \ell\colon\ x = *p \mid \ell\colon\ *p = x \mid \\
& & \ell\colon\ p = \texttt{malloc} \mid \ell\colon\ \texttt{skip} \mid s';\ s'' \mid \\
& & \ell\colon\ \texttt{if (}x\texttt{)}\ s'\ \texttt{else}\ s'' \mid \ell\colon\ \texttt{while (}x\texttt{)\ \{}\ s'\ \texttt{\}}
\end{array}
$$

$s, s', s'' \in$ Statement $\quad f \in$ InputField

$x, y, z, p \in$ Var $\quad c \in$ Int $\quad \ell \in$ Label

Figure 3. The core programming language

Figure 3 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 $\ell\colon\ x = y\ op\ z$ (where x, y, and z are either non-aliased variables or automatically generated temporaries, *op* represents binary arithmetic operations) and 2) all accesses to potentially aliased memory locations occur in load or store statements of the form $\ell\colon\ x = *p$ or $\ell\colon\ *p = x$. Each statement contains a unique label $\ell \in$ Label.

A statement of the form "$\ell\colon\ x = \texttt{read(}f\texttt{)}$" reads a value from an input field f. Because the input may contain multiple instantiations of the field f, different executions of the statement may return different values. For example, the loop at lines 14–17 in Figure 1 reads multiple instantiations of the `h_sample` and `v_sample` input fields.

Labels and Pointer Analysis: Figure 4 presents four utility functions 1) first_S : Statement $\rightarrow$ Statement, 2) first_L : Statement $\rightarrow$ Label, 3) last : Statement $\rightarrow$ Label, and 4) labels : Statement $\rightarrow$ Label in our notations. Given a statement s, $\text{first}_S(s)$ maps s to the first atomic statement inside s, $\text{first}_L(s)$ maps s to the label that corresponds to the first atomic statement inside s, $\text{last}(s)$ maps s to the label that corresponds to the last atomic statement inside s, and $\text{labels}(s)$ maps s to the set of labels that are inside s.

$$
\text{first}_S(s) = \begin{cases} \text{first}_S(s') & s = s';\ s'' \\ s & \text{otherwise} \end{cases}
$$

$$
\text{first}_L(s) = \text{the label of } \text{first}_S(s)
$$

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

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

Figure 4. Definitions of first_S, first_L, 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 the DSA pointer analysis [20] in combination with the basic pointer analysis pass in LLVM [2] to disambiguate aliases at load and store statements. Our underlying pointer analysis [2, 20] provides two functions no_alias and must_alias:

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

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

$$
\begin{array}{rcl}
C & := & C \wedge \text{safe}(e) \mid \text{safe}(e) \\
e & := & e'\ op\ e'' \mid atom \\
atom & := & x \mid c \mid f\langle id\rangle \mid \ell\langle id\rangle
\end{array}
$$

$id \in \text{Index} = \{1, 2, \ldots\}$ $\quad x \in$ Var $\quad c \in$ Int

$\ell \in$ LoadLabel $\quad f \in$ InputField

Figure 5. The condition syntax

Condition Syntax: Figure 5 presents the definition of *symbolic conditions* that our analysis manipulates and propagates. A condition C consists of a set of conjuncts of the form $\text{safe}(e)$. Each conjuct $\text{safe}(e)$ requires that the evaluation of the *symbolic expression* e (including all subcomputations in the evaluation, see Section 4.5) should not trigger an overflow.

Symbolic conditions C may contain four kinds of atoms: c represents a constant, x represents the variable x, $f\langle id\rangle$ represents a value from the input field f, and $\ell\langle id\rangle$ represents a value returned by the load statement with the label ℓ.

Abstraction for Input Field Instantiations: An atom $f\langle id\rangle$ in a symbolic condition C represents the value of an arbitrary instantiation of the input field f. The analysis uses the natural number id to distinguish values of potentially different instantiations of f. So all occurrences of a given atom $f\langle id\rangle$ in a symbolic condition C represent the same value. If $id_1 \neq id_2$, then $f\langle id_1\rangle$ and $f\langle id_2\rangle$ may

$$
\begin{array}{llll}
F(\ell\texttt{:}\ \ x\texttt{ = }c & , \ell & , C) = C[c/x] \\
F(\ell\texttt{:}\ \ x\texttt{ = }y & , \ell & , C) = C[y/x] \\
F(\ell\texttt{:}\ \ x\texttt{ = }y\ op\ z & , \ell & , C) = C[y\ op\ z/x] \\
F(s'\texttt{; }s'' & , \ell' \in \mathrm{labels}(s') & , C) = F(s', \ell', C) \\
F(s'\texttt{; }s'' & , \ell'' \in \mathrm{labels}(s'') & , C) = F(s', \mathrm{last}(s'), F(s'', \ell'', C)) \\
F(\ell\texttt{:}\ \ \texttt{if (}v\texttt{) }s'\texttt{ else }s'' & , \ell & , C) = F(s', \mathrm{last}(s'), C) \wedge F(s'', \mathrm{last}(s''), C) \\
F(\ell\texttt{:}\ \ \texttt{if (}v\texttt{) }s'\texttt{ else }s'' & , \ell' \in \mathrm{labels}(s') & , C) = F(s', \ell', C) \\
F(\ell\texttt{:}\ \ \texttt{if (}v\texttt{) }s'\texttt{ else }s'' & , \ell'' \in \mathrm{labels}(s'') & , C) = F(s'', \ell'', C) \\
F(\ell\texttt{:}\ \ \texttt{while (}v\texttt{) \{ }s'\texttt{ \}} & , \ell & , C) = C_{\text{fix}} \wedge C \quad \text{where } C_{\text{fix}} = \mathrm{norm}(F(s', \mathrm{last}(s'), C_{\text{fix}} \wedge C)) \\
F(\ell\texttt{:}\ \ \texttt{while (}v\texttt{) \{ }s'\texttt{ \}} & , \ell' \in \mathrm{labels}(s') & , C) = F(s, \ell, F(s', \ell', C)) \\
F(\ell\texttt{:}\ \ p\texttt{ = malloc} & , \ell & , C) = C \\
F(\ell\texttt{:}\ \ x\texttt{ = read(}f\texttt{)} & , \ell & , C) = C[f\langle id\rangle/x] \quad \text{where } id \text{ is fresh} \\
F(\ell\texttt{:}\ \ x\texttt{ = *}p & , \ell & , C) = C[\ell\langle id\rangle/x] \quad \text{where } id \text{ is fresh} \\
F(\ell\texttt{:}\ \ \texttt{*}p\texttt{ = }x & , \ell & , C) = C(\ell_1\langle id_1\rangle, \ell, x)(\ell_2\langle id_2\rangle, \ell, x) \cdots (\ell_n\langle id_n\rangle, \ell, x) \text{ for all } \ell_1\langle id_1\rangle, \cdots, \ell_n\langle id_n\rangle \text{ in } C
\end{array}
$$

$$
\text{where } C(\ell_i\langle id_i\rangle, \ell, x) = \begin{cases} C & \text{if no\_alias}(\ell, \ell_i) \\ C[x/\ell_i\langle id_i\rangle] & \text{if } \neg\,\text{no\_alias}(\ell, \ell_i) \wedge \text{must\_alias}(\ell, \ell_i) \\ C[x/\ell_i\langle id_i\rangle] \wedge C & \text{if } \neg\,\text{no\_alias}(\ell, \ell_i) \wedge \neg\,\text{must\_alias}(\ell, \ell_i) \end{cases}
$$

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

represent different instantiations of f (and may therefore represent different values).

Abstraction for Values Accessed via Pointers: An atom $\ell\langle id\rangle$ in a symbolic condition C represents an arbitrary value returned by the load statement at the label ℓ. The analysis uses the natural number id to distinguish potentially different values loaded at different executions of the load statement. Our analysis materializes atoms of the form $\ell\langle id\rangle$ in the propagated symbolic condition when it analyzes the load statement at the label ℓ. The analysis will eventually replace these atoms with appropriate expressions based on the aliasing information when it analyzes store statements that may store the corresponding value from the previously analyzed load statement at ℓ. In our abstraction, $\ell\langle id\rangle$ represents an *arbitrary* value that may be stored via any alias of the pointer dereferenced at ℓ during the execution from the starting point of the program to the current program point of the propagated symbolic condition C along any possible execution path (see Section 4.5).

Analysis Framework: Given a sequence of statements s, a label ℓ within s ($\ell \in \mathrm{labels}(s)$), and a symbolic condition C at the program point *after* the corresponding statement at the label ℓ, our demand-driven backwards analysis computes a symbolic condition $F(s, \ell, C)$. The analysis ensures that if $F(s, \ell, C)$ holds before executing s, then C will hold whenever the execution reaches the program point *after* the corresponding statement at the label ℓ (see Section 4.6 for the formal definition).

Given a program s_0 as a sequence of statements and a variable v at a critical site associated with the label ℓ, our analysis generates the condition $F(s_0, \ell, \mathrm{safe}(v))$ 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 $f\langle id\rangle$, depending on the assignment statement). Here the notation $C[e'/e]$ denotes the new symbolic condition obtained by replacing every occurrence of e in C with e'. The analysis rule for the input read statement materializes a new id to represent the read value $f\langle id\rangle$, because the variable x may get the value of a fresh instantiation of the input field f after the statement. This mechanism enables the analysis to correctly distinguish potentially different instantiations of the same input field (because values from potentially different instantiations 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 sequences of statements propagates the symbolic condition 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 $\ell\langle id\rangle$ 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 $\ell\langle id\rangle$ 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 $\ell\langle id\rangle$ 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 $\ell\langle id\rangle$ with x. If the alias analysis determines that the $\ell\langle id\rangle$ 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 $\ell\langle id\rangle$ replaced by x (for executions in which the load statement loads x) and one that leaves $\ell\langle id\rangle$ 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 [20], 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 13 in Section 5.2).

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 statement `while (`x`) { `s'` }` computes a sequence of symbolic conditions C_i, where $C_0 = \emptyset$ and $C_i = \mathrm{norm}(F(s', \mathrm{last}(s'), C \wedge C_{i-1}))$. Conceptually,

Input : original expression e_{orig}
Output: normalized expression e_{norm}

$e_{\text{norm}} \longleftarrow e_{\text{orig}}$
$counter_{\text{field}} \longleftarrow \{\_ \mapsto 0\}$
$counter_{\text{label}} \longleftarrow \{\_ \mapsto 0\}$
for a **in** $\text{atoms}(e_{\text{orig}})$ **do**
  if *a is in form $f\langle id\rangle$* **then**
    $n' \longleftarrow counter_{\text{field}}(f) + 1$
    $counter_{\text{field}} \longleftarrow counter_{\text{field}}[f \mapsto n']$
    $e_{\text{norm}} \longleftarrow e_{\text{norm}}[*f\langle n'\rangle / f\langle id\rangle]$
  else if *a is in form $\ell\langle id\rangle$* **then**
    $n' \longleftarrow counter_{\text{label}}(\ell) + 1$
    $counter_{\text{label}} \longleftarrow counter_{\text{label}}[\ell \mapsto n']$
    $e_{\text{norm}} \longleftarrow e_{\text{norm}}[*\ell\langle n'\rangle / \ell\langle id\rangle]$
for a **in** $\text{atoms}(e_{\text{norm}})$ **do**
  if *a is in form $*f\langle id\rangle$* **then**
    $e_{\text{norm}} \longleftarrow e_{\text{norm}}[f\langle id\rangle / *f\langle id\rangle]$
  else if *a is in form $*\ell\langle id\rangle$* **then**
    $e_{\text{norm}} \longleftarrow e_{\text{norm}}[\ell\langle id\rangle / *\ell\langle id\rangle]$

Figure 7. Normalization function $\text{norm}(e)$. $\text{atoms}(e)$ is a list that iterates over the distinct atoms in the expression e from left to right in order. The pseudo-code introduces temporary atoms of the forms "$*f\langle id\rangle$" and "$*\ell\langle id\rangle$" to avoid conflicts with existing original atoms in e_{norm}.

$F(\ell\texttt{:}\ v\ \texttt{=}\ \texttt{call}\ proc\ v_1 \cdots v_k, \ell, C)$
 where $proc \equiv proc(a_1, a_2, \cdots, a_k)\ \texttt{\{}\ s\texttt{;}\ \texttt{return}\ v_{ret}\ \texttt{\}}$ and
   $\ell_1\langle id_1\rangle, \cdots, \ell_n\langle id_n\rangle$ are all atoms of the form $\ell\langle id\rangle$ in C
begin
  $C' \longleftarrow \emptyset$
  $C_v \longleftarrow F(s, \text{last}(s), \text{safe}(v_{ret}))$
  for e_v **in** $\text{exprs}(C_v[v_1/a_1]\cdots[v_k/a_k])$ **do**
    $C_1 \longleftarrow F(s, \text{last}(s), \text{safe}(\ell_1\langle id_1\rangle))$
    for e_1 **in** $\text{exprs}(C_1[v_1/a_1]\cdots[v_k/a_k])$ **do**
      $\cdots$
      $C_n \longleftarrow F(s, \text{last}(s), \text{safe}(\ell_n\langle id_n\rangle))$
      for e_n **in** $\text{exprs}(C_n[v_1/a_1]\cdots[v_k/a_k])$ **do**
        $e'_v \longleftarrow \text{make\_fresh}(e_v, C)$
        $e'_1 \longleftarrow \text{make\_fresh}(e_1, C)$
        $\cdots$
        $e'_n \longleftarrow \text{make\_fresh}(e_n, C)$
        $C' \longleftarrow$
        $C' \wedge C[e'_v/v][e'_1/\ell_1\langle id_1\rangle]\cdots[e'_n/\ell_n\langle id_n\rangle]$
  return C'

Figure 8. Procedure call analysis algorithm. $\text{exprs}(C)$ returns the set of expressions that appear in the conjuncts of C. For example, $\text{exprs}(\text{safe}(e_1) \wedge \text{safe}(e_2)) = \{e_1, e_2\}$. $\text{make\_fresh}(e, C)$ renumbers ids in e so that atoms of the forms $\ell\langle id\rangle$ and $f\langle id\rangle$ will not conflict with existing atoms in C.

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 $F(s', \text{last}(s'), C \wedge C_{i-1})$ after each iteration. For a symbolic condition $C = \text{safe}(e_1) \wedge \cdots \wedge \text{safe}(e_n)$, the normalization of C is $\text{norm}(C) = \text{remove\_dup}(\text{safe}(\text{norm}(e_1)) \wedge \cdots \wedge \text{safe}(\text{norm}(e_n)))$, where $\text{norm}(e_i)$ is the normalization of each individual expression in C (using the algorithm presented in Figure 7) and $\text{remove\_dup}()$ 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 $f\langle id\rangle$ and $\ell\langle id\rangle$ with fresh id's. The materialized $f\langle id\rangle$ represent values of input field instantiations that the current loop iteration reads; the materialized $\ell\langle id\rangle$ 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. This normalization enables the analysis to recognize loop invariants that show up as equivalent successive analysis results that differ only in the materialized id's that they use to represent input field instantiations and values accessed via pointers.

The normalization algorithm is sound because 1) all occurrences of $f\langle id_1\rangle, ..., f\langle id_k\rangle$ are interchangeable, and 2) the normalization only renumbers the $id_1, ..., id_k$. The normalized condition is therefore equivalent to the original condition (see Section 4.6).

The normalization 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 sites in Figure 1 — 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.

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 Interprocedural Analysis

Analyzing Procedure Calls: Figure 8 presents the interprocedural analysis for procedure call sites. Given a symbolic condition C and a function call statement $\ell\texttt{:}\ v\ \texttt{=}\ \texttt{call}\ proc\ v_1 \cdots v_k$ that invokes a procedure $proc(a_1, a_2, \cdots, a_k)\ \texttt{\{}\ s\texttt{;}\ \texttt{return}\ v_{ret}\ \texttt{\}}$, the analysis computes $F(\ell\texttt{:}\ v\ \texttt{=}\ \texttt{call}\ proc\ v_1 \cdots v_k, \ell, C)$.

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 $\ell\langle id\rangle$ 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 $\ell\langle id\rangle$ 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 6) or the value of $\ell\langle id\rangle$ (see lines 8 and 11). The analysis avoids redundant analysis of the invoked procedure by caching the analysis results $F(s, \text{last}(s), \text{safe}(v_{ret}))$ and $F(s, \text{last}(s), \text{safe}(\ell\langle id\rangle))$ 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 7, 9, and 12 in Figure 8).

Also note that the analysis renumbers the ids in the symbolic expressions derived from an analysis of the invoked procedure before the replacements (see lines 13–16). This ensures that the atoms of the forms $f\langle id\rangle$ and $\ell\langle id\rangle$ in the expressions are fresh and will not conflict with existing atoms in C after replacements.

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) [3], 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`) as critical sites. SIFT can also be configured to recognize additional memory allocation and block copy routines (for example, `dMalloc` in Dillo).

Bit Width and Signedness: 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 [20] 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. SIFT currently contains predefined annotations for a small set of important standard library functions that influence our integer overflow analysis (e.g., `memset`() and `strlen`()).

If both the source code and the annotation for an invoked procedure are 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.

Command Line Arguments: At four of the 56 critical sites in our analyzed benchmark modules, the absence of overflow depends, in part, on the lengths of the command line arguments. The generated final symbolic condition that SIFT uses to generate the input filter therefore contains variables that represent these lengths. Our currently implemented system sets these lengths to a specific constant value greater than the maximum length of the command line arguments of all benchmark applications. In production use, we expect SIFT deployments to either 1) check that the command line argument lengths are less than this constant value before launching the application or 2) dynamically extract the command line argument lengths when the application is launched, then provide these lengths to the filter.

Annotations for Input Read Statements: 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 1 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:

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

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).

3.5 Input Filter Generation

The SIFT filter generator prunes any conjuncts that contain residual occurrences of abstract materialized values $\ell\langle id\rangle$ in the final symbolic condition C. It also replaces every residual occurrence of program variables v with 0. These residual occurrences correspond to initial values in the program state $\overline{\sigma}$ and $\overline{h}$ in the abstract semantics (see Section 4.3). After pruning, the final condition C_{Inp}

contains only input field variables of the form $f\langle id\rangle$ and constant atoms.

In effect, the pruning algorithm eliminates any checks involving uninitialized data from the filter — SIFT filters are not designed to nullify overflow errors that may occur when the program accesses uninitialized data (which, in C, may contain arbitrary values). The SIFT soundness theorem (Theorem 4) reflects this restriction. For languages such as Java which initialize data to specific values, the SIFT filter generator would not prune conjuncts involving references to uninitialized data. It would instead protect against overflows involving uninitialized data by replacing residual occurrences of abstract materialized values and program variables with the corresponding initial values.

The generated 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 range 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 11 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 parsed input against all final symbolic conditions in sequence. 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 14) over all of the final symbolic condition checks.

4. Soundness of the Static Analysis

We next formalize our static analysis algorithm on the core language in Figure 3 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 $(\sigma, \rho, \varsigma, \varrho, \mathit{Inp})$ as follows:

$$\begin{array}{ll}\sigma\colon \text{Var} \to (\text{Loc}+\text{Int}+\{\text{undef}\}) & \varsigma\colon \text{Var} \to \text{Bool}\\ \rho\colon \text{Loc} \to (\text{Loc}+\text{Int}+\{\text{undef}\}) & \varrho\colon \text{Loc} \to \text{Bool}\\ \mathit{Inp}\colon \text{InputField} \to \mathcal{P}(\text{Int}) & \end{array}$$

σ and ρ map variables and memory locations to their corresponding values. We use undef to represent uninitialized values. We define that if any operand of an arithmetic operation is undef, the result of the operation is also undef. $\mathit{Inp}(f)$ maps the input field f to the corresponding set of the values of all instantiations of the field f in the input file. Inp therefore represents the input file, which remains 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 subcomputations) generates an overflow. ϱ maps each memory location to a boolean overflow flag similar to ς.

In the initial state $(\sigma_0, \rho_0, \varsigma_0, \varrho_0, \mathit{Inp})$, σ_0 and ρ_0 map all variables and locations to undef. 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 9 presents the small step dynamic semantics of the language. Note that in Figure 9, $\text{overflow}(a, b, op)$ 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 binop rule in Figure 9 appropriately updates the overflow flag of x in ς by checking whether the computation that generates the value of x (including the subcomputations that generates the value of y and z) results in an overflow condition.

Also note that the rule for the input read statement nondeterministically updates the value of x with an arbitrary element chosen from the set $\mathit{Inp}(f)$, which contains the values of all instantiations of the input field f in the input file. This semantics conservatively models the behavior of input read statements in C programs.

4.2 Soundness of the Pointer Analysis

Our analysis uses an underlying pointer analysis [20] to analyze programs that use pointers. The underlying pointer analysis provides two functions no_alias and must_alias to our main analysis. We formally state our assumptions about the soundness of the underlying pointer alias analysis as follows:

Definition 1 (no_alias and must_alias)**.** *Given any execution sequence*

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

*and two statements s_{store} (ℓ_{store} : *p = x) and s_{load} (ℓ_{load} : x′ = *p′), we have:*

$$\text{no\_alias}(\ell_{store}, \ell_{load}) \implies \forall i < j :\\ (\text{first}_L(s_i) = \ell_{store} \wedge \text{first}_L(s_j) = \ell_{load}) \to \sigma_i(p) \neq \sigma_j(p')$$

$$\text{must\_alias}(\ell_{store}, \ell_{load}) \implies \forall i < j :\\ (\text{first}_L(s_i) = \ell_{store} \wedge \text{first}_L(s_j) = \ell_{load} \wedge (\textstyle\bigwedge_{k\in(i,j)} \text{first}_L(s_k) \neq \ell_{store}))\\ \to (\sigma_i(p) = \sigma_j(p'))$$

4.3 Abstract Semantics

We next define an abstract semantics that allows us to prove the soundness of our static analysis algorithm. There are two key differences between the abstract and original semantics. First, for if and while statements, the abstract semantics conservatively ignores the condition and nondeterministically executes one of the two control flow branches. Second, the abstract semantics conservatively groups values that load and store statements access via pointers into equivalence classes based on the aliasing information from the underlying pointer or alias analysis. It then conservatively models reads via pointers as nondeterministically returning an arbitrary stored value from the corresponding equivalence class. We adopt this abstract semantics because it more closely reflects how SIFT incorporates the underlying pointer or alias analysis and analyzes if, while, store, and load statements.

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

$$\begin{array}{ll}\overline{\sigma}\colon \text{Var} \to \text{Int} & \overline{\varsigma}\colon \text{Var} \to \text{Bool}\\ \overline{\hbar}\colon \text{LoadLabel} \to \mathcal{P}(\text{Int} \times \text{Bool}) & \end{array}$$

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.

In the initial state $(\overline{\sigma}_0, \overline{\varsigma}_0, \overline{\hbar}_0, \mathit{Inp})$, $\overline{\sigma}_0$ and $\overline{\varsigma}_0$ map all variables to 0 and false respectively. $\overline{\hbar}_0$ maps all labels of load statements to the empty set.

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 9. The small step operational semantics of the language. "nil" is a special label reserved by the semantics.

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

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

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

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

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

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

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

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

Small Step Rules: Figure 10 presents the small step rules for the abstract semantics. We omit rules for simple assignment statements, while statements, and sequence statements for brevity.

The rules for if, while, malloc, load, and store statements reflect the primary differences between the abstract and original semantics. The rules for if and while statements (if-t, if-f, and the omitted while statement rules) in the abstract semantics conservatively ignore the condition and nondeterministically execute one of the two control flow branches. The rule for store statements maintains the state $\overline{\hbar}$ according to the aliasing information. The rule for load statements nondeterministically returns an element from the corresponding set in $\overline{\hbar}$.

4.4 Relationship of the Original and the Abstract Semantics

We formally state the relationship between the original and abstract semantics as follows.

Theorem 2. *For any execution trace in the original semantics:*

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

there is an execution trace in the abstract semantics:

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

such that the following conditions hold:

$$\forall i\, \forall x \in Var : \sigma_i(x) \notin Int \vee (\sigma_i(x) = \overline{\sigma}_i(x) \wedge \varsigma_i(x) = \overline{\varsigma}_i(x))$$

$$\forall i : \text{first}_S(s_i) = \text{“}\ell :\ x = *p\text{”} \rightarrow (\rho_i(\sigma_i(p)) \notin Int \vee \langle \rho_i(\sigma_i(p)), \varrho_i(\sigma_i(p))\rangle \in \overline{\hbar}_i(\ell))$$

The intuition behind the first condition 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 behind the second condition is that $\overline{\hbar}_i(\ell)$ corresponds to the possible values that the corresponding load statement of the label ℓ may obtain from the memory. When a load statement executes in the original semantics, the obtained integer value is in the corresponding set in $\overline{\hbar}_i$ in the abstract semantics.

This theorem connects an arbitrary program execution in the original semantics to a corresponding execution in the abstract semantics. An important consequence of this theorem is that the soundness of our analysis in the abstract semantics implies the soundness of the analysis in the original semantics. See our technical report [23] for the proof sketch of this theorem.

4.5 Evaluation of the Symbolic Condition

Our static analysis maintains and propagates a symbolic condition C. Figure 11 defines the evaluation rules of the symbolic condition C over an abstract program state $(\overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp)$ The notation $(\overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp) \models C$ denotes that the abstract program state $(\overline{\sigma}, \overline{\varsigma}, \overline{\hbar}, Inp)$ satisfies the condition C. The evaluation rule for $\text{safe}(e_1\ op\ e_2)$ checks that no overflow error occurs in any subcomputation that contributes to the final value of $(e_1\ op\ e_2)$.

In our abstraction, each atom of the form $f\langle id\rangle$ corresponds to the value of an arbitrary instantiation of the input field f – i.e., $f\langle id\rangle$ corresponds to an arbitrary element of the set $Inp(f)$. Each atom of the form $\ell\langle id\rangle$ corresponds to an arbitrary value that may be stored via any alias of the corresponding pointer – i.e., $\ell\langle id\rangle$

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

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

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

Figure 11. Symbolic condition evaluation rules.

corresponds to an arbitrary element of the set $\overline{\hbar}(\ell)$. The evaluation rules enumerate all possible bindings of $f\langle id\rangle$ and $\ell\langle id\rangle$ to check that no binding causes an overflow.

The definition of the evaluation rules also ensures that all occurrences of $f\langle id\rangle$ which reference the same input field f and all occurrences of $\ell\langle id\rangle$ materialized from the same load statement ℓ are interchangable. This interchangability ensures that the normalization algorithm in Section 3.2 is sound — i.e., that renumbering ids in a symbolic condition C does not change the meaning of the condition. Therefore, given symbolic condition C, the normalization algorithm produces an equivalent condition $\text{norm}(C)$.

4.6 Soundness of the Analysis

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 $\ell \in \text{labels}(s_i)$ and a start condition C, our analysis generates a condition $F(s_i, \ell, C)$, such that if $(\overline{\sigma}_i, \overline{\varsigma}_i, \overline{\hbar}_i, Inp) \models F(s_i, \ell, C)$, then*

$$\begin{array}{l}((\langle s_i, \overline{\sigma}_i, \overline{\varsigma}_i, \overline{\hbar}_i\rangle \longrightarrow_a^* \\ \langle s_{j-1}, \overline{\sigma}_{j-1}, \overline{\varsigma}_{j-1}, \overline{\hbar}_{j-1}\rangle \longrightarrow_a \\ \langle s_j, \overline{\sigma}_j, \overline{\varsigma}_j, \overline{\hbar}_j\rangle) \wedge (\text{first}_L(s_{j-1}) = \ell)) \\ \implies ((\overline{\sigma}_j, \overline{\varsigma}_j, \overline{\hbar}_j, Inp) \models C)\end{array}$$

This theorem guarantees that if the abstract program state before executing s_i satisfies $F(s_i, \ell, C)$, then the abstract program state at the program point after the statement at label ℓ will always satisfy C (here the notation " $\longrightarrow_a^*$ " denotes the execution of 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 semantics based on the soundness property over the abstract semantics:

Theorem 4. *Given a program s_0, a program point $\ell \in \text{labels}(s_0)$, and a program variable v, our analysis generates a condition $C = F(s_0, \ell, \text{safe}(v))$, such that if $(\overline{\sigma}_0, \overline{\varsigma}_0, \overline{\hbar}_0, Inp) \models C$, then*

$$\begin{array}{l}((\langle s_0, \sigma_0, \rho_0, \varsigma_0, \varrho_0\rangle \longrightarrow^* \\ \langle s_{n-1}, \sigma_{n-1}, \rho_{n-1}, \varsigma_{n-1}, \varrho_{n-1}\rangle \longrightarrow \\ \langle s_n, \sigma_n, \rho_n, \varsigma_n, \varrho_n\rangle) \wedge (\text{first}_L(s_{n-1}) = \ell)) \\ \implies (\sigma_n(v) \notin \text{Int} \vee \varsigma_n(v) = \text{false})\end{array}$$

This theorem guarantees that if the input satisfies the generated condition C (note that $(\overline{\sigma}_0, \overline{\varsigma}_0, \overline{\hbar}_0, Inp)$ is the predefined constant initial state in Section 4.3), then for any execution in the original semantics (here the notation " $\longrightarrow^*$ " denotes the execution of 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 uninitialized access), the computation history for obtaining this integer value contains no overflow error.

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

Figure 12. 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.

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 [15]). All experiments were conducted on an Intel Xeon X5363 3.00GHz machine running Ubuntu 12.04.

We focus on media and browser applications (client applications) in our experiments due to the ease of obtaining inputs required to evaluate our system against false positives. Other types of applications (e.g., server applications) would require additional infrastructure (i.e., recording network traffic) to test. By design, SIFT is applicable to a wide range of applications.

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 56 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).

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 12 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

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 13. Static analysis and filter generation results

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

Analysis Evaluation: Figure 13 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. The sizes at 29 of the 58 sites depend on values computed inside loops. To generate input filters for these sites, SIFT must therefore compute loop invariants that capture the effect of the loop on the sizes that occur at these sites.

The seventh column (Max Condition Size) presents, for each application module, the maximum number of conjuncts 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.

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 14. Generated filter results.

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 two of the 58 call sites (one in Swfdec and one in png2swf). The expressions at these two sites contain subexpressions whose values depend on an unbounded number of values computed in loops. 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.

Filter Evaluation: 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 14 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 Filter Confirmation on Vulnerabilities

Each benchmark application contains a known integer overflow vulnerability. To confirm that the generated filters operate correctly, we obtained, for each vulnerability, a malicious input that triggers the integer overflow error. We confirmed that the filters successfully identified and discarded all of these malicious inputs. We also manually examined the root cause of each vulnerability and confirmed that the generated filters completely nullified the vulnerability — if an input passes the filter, it will not trigger the overflow error that enables the vulnerability. See our technical report [23] for the detailed case study of each vulnerability.

5.4 Discussion

The experimental results highlight the combination of properties that, together, enable SIFT to effectively nullify 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 Analysis: Madhavan et. al. present an approximate weakest precondition analysis to verify the absence of null dereference errors in Java programs [24]. The goal is to verify that, for each pointer dereference, there are appropriate null checks within the program that guard the dereference to ensure that the program will never execute the dereference with a null pointer value. The analysis tracks only pointer dereferences, pointer assignments, and conditions involving null pointer checks.

SIFT faces different challenges and therefore uses different techniques. Instead of using a finite domain to track dereferenced pointers and pointer assignments, SIFT must track potentially unbounded arithmetic symbolic expressions involving input fields and values accessed via pointers. To successfully analyze loops, SIFT's abstraction and expression normalization algorithms work together to discover invariants for loops that may access a statically unbounded number of input field instantiaions (as long as the value of the analyzed expression depends only on a statically bounded number of input field instantiations).

Flanagan et. al. present a general intraprocedural weakest precondition analysis for generating verification conditions for ESC/JAVA programs [14]. 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 [16, 19, 34]. Two key differences are that SIFT statically analyzes the application, not its inputs, and takes all execution paths into account to generate a sound filter.

Input Rectification: Input rectification [22, 30] empirically learns input constraints from benign training inputs. It then monitors inputs for violations of the learned constraints. Instead of discarding inputs that violate the learned constraints, input rectification modifies the input so that it satisfies the constraints. The goal is to nullify potential errors while still enabling the program to successfully process as much input data as possible. Because it learns the constraints from examples, this technique is not sound — the generated filter may miss some inputs that target the error. It would be possible to combine SIFT with input rectification to obtain a sound input rectification technique.

Runtime Repair: Researchers have developed a range of techniques for dynamically detecting and repairing errors in the program's execution [18, 27–29]. These techniques all purposefully change the program's semantics in an attempt to enable the program to successfully process inputs that it would otherwise be unable to process without error. The goal of SIFT, in contrast, is to nullify errors without changing the program's semantics by discarding inputs that might trigger the errors.

Static Analysis for Finding Integer Overflow and Sign Errors: Several static analysis tools have been proposed to find integer overflow and/or sign errors [7, 31, 35]. KINT [35], 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 [35].

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 Bug Finding and Validation: DART [17] and KLEE [6] use symbolic execution to automatically generate test inputs that systematically exercise different control-flow paths in a program. One goal is to find inputs that expose errors in the program. IntScope [33] and SmartFuzz [25] are symbolic execution systems designed specifically to expose integer overflow and/or sign 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.

Snugglebug [8] is a backward symbolic analysis engine for error validation — the goal is to generate an input that can trigger a potential error identified by some other means. As with other symbolic execution systems, Snugglebug is designed to enumerate all potential paths that the program may take to reach the error. If the number of paths is large or unbounded (for example, if the program contains loops), it may be infeasible to enumerate all paths. SIFT, in contrast, uses a precondition generation technique that can successfully analyze large programs while soundly taking all paths into consideration.

Runtime Checks 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 [4, 9, 13, 37]. Another technique is to use safe integer libraries such as SafeInt [21] and CERT's IntegerLib [32] to perform sanity checks at runtime. Using these libraries requires developers to rewrite existing code to use safe versions of integer operations.

However, the inserted code typically imposes non-negligible overhead. When they detect an error, these techniques typically generate a warning and terminate the execution (effectively turning any integer overflow attack into a denial of service attack). SIFT, in contrast, inserts no code into the application and nullifies integer overflow errors by discarding inputs that trigger such errors.

Benign Integer Overflows: In some cases, developers may intentionally write code that contains benign integer overflows [13, 33, 35]. A potential concern is that techniques that nullify overflows may interfere with the intended behavior of such programs [13, 33, 35]. Because SIFT focuses on critical memory allocation and block copy sites that are unlikely to have such intentional integer overflows, it 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 sound, efficient, and precise input filters for the vast majority of memory allocation and block copy call sites in our analyzed benchmark modules.

Acknowledgements

We thank Michael Carbin, Sasa Misailovic, and the anonymous reviewers for their insightful comments. We note our earlier technical report [23]. This research was supported by DARPA (Grant FA8650-11-C-7192).

References

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

[2] LLVM Basic Alias Analysis Pass. `http://llvm.org/docs/AliasAnalysis.html#the-basicaa-pass`.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 363–374, New York, NY, USA, 2009. ACM.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] M. Kling, S. Misailovic, M. Carbin, and M. Rinard. Bolt: on-demand infinite loop escape in unmodified binaries. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12', pages 431–450, New York, NY, USA, 2012. ACM.

[19] 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.

[20] 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.

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

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

[23] F. Long, S. Sidiroglou, D. Kim, and M. Rinard. Sound input filter generation for integer overflow errors. MIT-CSAIL-TR-2013-018.

[24] 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.

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

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

[27] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: Automatically correcting memory errors with high probability. In *In Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM*. Press, 2007.

[28] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 87–102, New York, NY, USA, 2009. ACM.

[29] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *In Proceedings 6 th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 303–316, 2004.

[30] M. C. Rinard. Living in the comfort zone. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07. ACM, 2007.

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

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

[33] 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.

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

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

[36] 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.

[37] 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.