Proving Memory Safety of Floating-Point Computations by Combining Static and Dynamic Program Analysis

Patrice Godefroid
Microsoft Research
Redmond, WA, USA
pg@microsoft.com

Johannes Kinder [*]
Technische Universität Darmstadt
Darmstadt, Germany
kinder@cs.tu-darmstadt.de

Abstract

Whitebox fuzzing is a novel form of security testing based on runtime symbolic execution and constraint solving. Over the last couple of years, whitebox fuzzers have found dozens of new security vulnerabilities (buffer overflows) in Windows and Linux applications, including codecs, image viewers and media players. Those types of applications tend to use floating-point instructions available on modern processors, yet existing whitebox fuzzers and SMT constraint solvers do not handle floating-point arithmetic. Are there new security vulnerabilities lurking in floating-point code?

A naive solution would be to extend symbolic execution to floating-point (FP) instructions (months of work), extend SMT solvers to reason about FP constraints (months of work), and then face more complex constraints and an even worse path explosion problem. Instead, we propose an alternative approach, based on the rough intuition that FP code should only perform memory-safe data-processing of the "payload" of an image or video file, while the non-FP part of the application should deal with buffer allocations and memory address computations, with only the latter being prone to buffer overflows and other security critical bugs. Our approach combines (1) a lightweight local path-insensitive "may" static analysis of FP instructions with (2) a high-precision whole-program path-sensitive "must" dynamic analysis of non-FP instructions. The aim of this combination is to prove memory safety of the FP part of each execution and a form of *non-interference* between the FP part and the non-FP part with respect to memory address computations.

We have implemented our approach using two existing tools for, respectively, static and dynamic x86 binary analysis. We present preliminary results of experiments with standard JPEG, GIF and ANI Windows parsers. For a given test suite of diverse input files, our mixed static/dynamic analysis is able to prove memory safety of FP code in those parsers for a small upfront static analysis cost and a marginal runtime expense compared to regular runtime symbolic execution.

1. Introduction

Whitebox fuzzing [10] is a promising new form of security testing based on dynamic test generation [3, 8]. Starting with a well-formed input, whitebox fuzzing executes the program under test while simultaneously performing symbolic execution to generate input constraints from conditional statements that capture the program execution path. Those constraints are then systematically negated and solved with a constraint solver, generating new test inputs to exercise different execution paths of the program. This process is repeated with the goal of exercizing many program paths and finding many bugs. Over the last couple of years, whitebox fuzzers have found dozens of new security vulnerabilities (buffer overflows) in Windows [10] and Linux [14] applications, including codecs, image viewers and media players. Notably, whitebox fuzzing was recently credited to have found roughly one third of all the bugs discovered by file-fuzzing during the development of Microsoft's Windows 7 [7].

Many image and video codecs use floating-point instructions available on modern processors. Yet existing whitebox fuzzers do not generate constraints for floating-point code, perhaps because current mainstream SMT solvers (such as STP, Yices, and Z3, to name a few) do not handle floating-point arithmetic.

In this paper, we present a new technique for proving memory safety of floating-point (FP) computations that does not require precise symbolic reasoning about FP code. The basic idea is to treat all FP values as a single special symbolic value "FP-tag" during symbolic execution, and to leverage existing symbolic evaluation rules to perform a dynamic taint-flow analysis of FP-tags. If an FP-tag is ever used to compute a memory address dereferenced during execution, a runtime error is generated as a warning.

Our intuition is that such warnings should be rare as floating-point computations should be involved *only* in the processing of the *"payload"* of an image or video input file, not in buffer allocations and memory address computations, with only the latter being prone to buffer overflows and other security critical bugs. Indeed, it would be intuitively surprising if processing, say, a JPEG image of a *white* flower would result in a memory safe execution, while processing the exact same image but with a *red* flower would trigger a buffer overflow. A related but simpler phenomenon is *data independence* in communication protocols [19]: only the control part (i.e.,

[*] The work of this author was done mostly while visiting Microsoft.

the header and footer) of a packet influences the behavior of protocol entity, while the packet content is just forwarded to a lower or higher protocol layer and is irrelevant to prove correctness. In contrast, image and media formats are usually much more complex, and the distinction between control and payload is made at runtime by treating differently different parts of the input. For instance, a movie file may start with a declaration that it contains 10,000 chunks of data. After this global header, a stream of all chunks follows, with each having its own private header describing whether it contains audio or video data, its size and encoding. Moreover, assuming we define the payload of a file as the bytes that are processed by FP code, payload data may influence the control flow of the image or movie processor whenever floating-point values are tested in conditional statements.

To prove memory safety of a conditional statement tainted by an FP-tag during symbolic execution, we use a lightweight static analysis for conservatively over-approximating *all* possible executions of the if-then-else block and then skip the entire block during symbolic execution, *provided* that statically-computed checks are satisfied at runtime. The satisfiability of those checks guarantee that all possible executions inside the block do not access any memory address computed using FP-tags or input values, and that all their side-effects are conservatively represented by new FP-tags injected when symbolic execution resumes at the end of the block.

We have implemented our approach using two existing tools for static and dynamic x86 binary analysis, respectively. We present preliminary results of experiments with standard JPEG, GIF and ANI processors embedded in Windows (and deployed on nearly a billion machines worldwide). For a given test suite of diverse input files, our mixed static/dynamic analysis is able to prove memory safety of FP code in the observed executions of those parsers for a small upfront static analysis cost and a marginal runtime expense compared to regular runtime symbolic execution, hence avoiding costly FP constraint generation and solving, and additional tests to cover FP code.

2. Background: Systematic Dynamic Test Generation

Dynamic test generation (see [8] for further details) consists of running the program P under test both concretely, executing the actual program, and symbolically, calculating constraints on values stored in program variables x and expressed in terms of input parameters. Side-by-side concrete and symbolic executions are performed using a concrete store M and a symbolic store $\mathcal{S}$, which are mappings from *memory addresses* (where program variables are stored) to concrete and symbolic values, respectively. A *symbolic value* is any expression e in some theory $\mathcal{T}$ where all free variables are exclusively input parameters. For any program variable x, $M(x)$ denotes the *concrete value* of x in M, while $\mathcal{S}(x)$ denotes the *symbolic value* of x in $\mathcal{S}$. For notational convenience, we assume that $\mathcal{S}(x)$ is always defined and is simply $M(x)$ by default if no expression in terms of inputs is associated with x in $\mathcal{S}$. When $\mathcal{S}(x)$ is different from $M(x)$, we say that the program variable x is "symbolic", meaning that its value is a function of some input(s), which is represented by the symbolic expression $\mathcal{S}(x)$ associated with x in the symbolic store. We also extend this notation to allow $M(e)$ to denote the concrete value of symbolic expression e when evaluated with the concrete store M. The notation + for mappings denotes updating; for example, $M' = M + [m \mapsto v]$ is the same map as M, except that $M'(m) = v$.

The program P manipulates the memory (concrete and symbolic stores) through statements or *commands*, which are abstractions of the machine instructions actually executed. A command can be an *assignment* of the form $v := e$ (where v is a program variable and e is an expression) or $[e_1] := e_2$ (where $[e_1]$ is a memory address dereference at the address defined by evaluating expression e_1, and e_2 is an expression), a *conditional statement* of the form `if` e `then` C' `else` C'' (where e denotes a Boolean expression, and C' and C'' denote the unique[1] next command to be evaluated when e holds or does not hold, respectively), or `stop` (which corresponds to a program error or normal termination).

Given an input vector I assigning a value I_i to the i-th input parameter, the evaluation of a program defines a unique finite[2] *program execution* $s_0 \xrightarrow{C_1} s_1 \ldots \xrightarrow{C_n} s_n$ that executes the finite sequence $C_1 \ldots C_n$ of commands and goes through the finite sequence $s_1 \ldots s_n$ of program states. Each *program state* is a tuple $\langle C, M, \mathcal{S}, pc\rangle$ where C is the next command to be evaluated, and pc is a special meta-variable that represents the current path constraint. For a finite sequence w of commands (i.e., a control path w), a *path constraint* pc_w is a formula of theory $\mathcal{T}$ that characterizes the input assignments for which the program executes along w. To simplify the presentation, we assume that all the program variables have some unique initial concrete value in the initial concrete store M_0, and that the initial symbolic store $\mathcal{S}_0$ identifies the program variables v whose values are program inputs (for all those, we have $\mathcal{S}_0(v) = I_i$ where I_i is the corresponding input parameter). Initially, pc is defined to `true`.

Systematic dynamic test generation [8] consists of systematically exploring all feasible program paths of the program under test by using path constraints and a constraint solver. By construction, a path constraint represents conditions on inputs that need to be satisfied for the current program path to be executed. Given a program state $\langle C, M, \mathcal{S}, pc\rangle$ and a constraint solver for theory $\mathcal{T}$, if C is a conditional statement of the form `if` e `then` C' `else` C'', any satisfying assignment to the formula $pc \wedge e$ (respectively $pc \wedge \neg e$) defines program inputs that will lead the program to execute the `then` (respectively `else`) branch of the conditional statement. By systematically repeating this process, such a *directed search* can enumerate all possible path constraints and eventually execute all feasible program paths.

The search is exhaustive provided that the generation of the path constraint (including the underlying symbolic execution) and the constraint solver for the given theory $\mathcal{T}$ are both *sound and complete*, that is, for all program paths w, the constraint solver returns a satisfying assignment for the path constraint pc_w *if and only if* the path is feasible (i.e., there exists some input assignment leading to its execution). In this case, in addition to finding errors such as the reachability of bad program statements (like `assert(false)`), a directed search can also prove their absence, and therefore obtain a form of program *verification*.

Theorem 1. *(adapted from [8]) Given a program P as defined above, a directed search using a path constraint generation and*

[1] We assume program executions are sequential and deterministic.

[2] We assume program executions terminate. In practice, a timeout prevents non-terminating program executions and issues a runtime error.

```
1  evalSymbolic($e$) =
2    match ($e$):
3      case $v$: // Program variable $v$
4        return $S(\&v)$
5      case +($e_1$, $e_2$): // Addition
6        $f_1$ = evalSymbolic($e_1$)
7        $f_2$ = evalSymbolic($e_2$)
8        if $f_1$ and $f_2$ are constants
9            return evalConcrete($e$)
10       else
11           return createExpression('+', $f_1$, $f_2$)
12     case [$e_1$]: // Address dereference
13       $c$ =evalConcrete($e_1$)
14       $mr_c$ =getMemoryRegion($c$)
15       if $mr_c$ is undefined // passive check
16           error('memory access violation at address $c$')
17       $f_1$ = evalSymbolic($e_1$)
18       if $f_1$ is not a constant
19           $pc = pc \wedge (0 \leq (f_1 - mr_c.base) \leq mr_c.size)$ // active check
20       return $S(c)$
21       etc.
```

Figure 1. Symbolic expression evaluation.

```
1  Procedure executeSymbolic($P$,$I$) =
2    initialize $M_0$ and $S_0$
3    path constraint $pc =$ true
4    $C$ = getNextCommand()
5    while ($C \neq$ stop)
6      match ($C$):
7        case ($v := e$):
8          $M = M + [\&v \mapsto$evalConcrete($e$)$]$
9          $S = S + [\&v \mapsto$evalSymbolic($e$)$]$
10       case ($[e_1] := e_2$):
11         $M = M + [$evalConcrete($e_1$) $\mapsto$evalConcrete($e_2$)$]$
12         evalSymbolic($[e_1]$) // passive and active check
13         $S = S + [$evalConcrete($e_1$) $\mapsto$evalSymbolic($e_2$)$]$
14       case (if $e$ then $C'$ else $C''$):
15         $b$ =evalConcrete($e$)
16         $c$ =evalSymbolic($e$)
17         if $b$ then $pc = pc \wedge c$
18         else $pc = pc \wedge \neg c$
19     $C$ = getNextCommand() // end of while loop
```

Figure 2. Symbolic execution.

a constraint solver that are both sound and complete exercises all feasible program paths exactly once.

In this case, if a program statement has not been executed when the search is over, this statement is not executable in any context. In practice, path constraint generation and constraint solving are usually not sound and complete. When a program expression cannot be expressed in the given theory $\mathcal{T}$ decided by the constraint solver, it can be simplified using concrete values of sub-expressions, or replaced by the concrete value of the entire expression.

Note that the above formalization and theorem do apply to programs containing loops or recursion, as long as all program executions terminate. However, in the presence of a single loop whose number of iterations depends on some unbounded input, the number of feasible program paths becomes infinite. In practice, termination can always be forced by bounding input values, loop iterations, or recursion, at the cost of potentially missing bugs.

Figure 1 illustrates how to symbolically evaluate expressions e involved in program instructions; Figure 2 shows how to generate a path constraint while symbolically executing a program. $\&v$ denotes the address at which the value of v is stored. As in [9], *all* symbolic expressions e ever used in the left hand-side (lines 9 and 13 of Figure 2) or right hand-side (line 12 of Figure 2) of an assignment statement or in the Boolean expression of a conditional statement (line 16 of Figure 2) are checked for memory access violations. Whenever a memory address is dereferenced during execution, an expression of the form $[e]$ is evaluated to compute that address. The concrete value c of the address is checked "passively" (in line 15 of Figure 1) to make sure it points to a valid memory region mr_c; then the symbolic expression e is also checked "actively" by injecting a new constraint in the path constraint (line 19 of Figure 1) to make sure other input values cannot trigger a buffer overflow or underflow at this point of the program execution [9]. How to keep track of the base address $mr_c.base$ and size $mr_c.size$ of each valid memory region mr_c during the execution of the program P is discussed in [5]. To simplify the presentation, Figure 1 only handles single symbolic pointer dereferences (by returning the value $S(c)$ in line 20), but could be extended to handle multiple levels of pointer indirections [5].

Given two program executions w and w', we write $w \equiv_P w'$ if they execute the same (finite) sequence of commands. Observe that $w \equiv_P w'$ implies $pc_w = pc_{w'}$ since both execute the same control path. Let an *extended path constraint epc* denote a path constraint extended with buffer-overflow checks injected as in line 19 of Figure 1. We write $w \equiv_{P+B} w'$ if $w \equiv_P w'$ *and* $epc_w = epc_{w'}$. Thus, two executions w and w' are equivalent with respect to $\equiv_{P+B}$ if they execute the same control path in the extended program $P + B$ which extends the original program P with bound checks for all memory accesses. Such bound checks are useful to prove memory safety.

Theorem 2. *(adapted from [9]) Given an extended program $P + B$ as defined above, a directed search using an extended path constraint generation and a constraint solver that are both sound and complete exercises all feasible program paths exactly once. Moreover, if no runtime error is ever generated by line 16 of Figure 1, all program executions are memory safe.*

3. Problem Definition

For *security analysis*, the advantages of binary analyses are well established: static [2,12] or dynamic [10,14] binary analyses allow to analyze the exact program that is being shipped to customers and that includes important details such as code transformations performed by compilers. In this work, we adopt a similar approach and build upon two existing tools for, respectively, static and dynamic x86 binary analysis of Windows applications. Specifically, the tool we extend for runtime symbolic execution handles a large set of x86 instructions, but no floating-point (FP) instructions.

We use the term *FP instructions* to refer to the about 100 floating point and over 300 SIMD instructions in the x86 architecture [11]. Floating point instructions execute on the x87 FPU, which used to be a separate physical unit on early x86 platforms. They primarily read from and write to the FPU's own register stack (`st(0)` to `st(7)`) and its status-, control-, and tag-registers. SIMD (Single Instruction, Multiple Data) instructions have been introduced into x86 as the MMX, 3DNow!, SSE, SSE2, SSE3, SSSE3, and SSE4 extensions [17]. These instructions also use special registers (`mmx0-7` as aliases of the FPU register stack, `xmm0-15`, and the control- and status register) and are specifically tailored towards fast vector operations. We refer to all registers used exclusively by floating point and SIMD instructions as *FP-registers*. Thus, the value stored in an FP-register can only be moved to memory or a regular register by an FP instruction.

For instance, here is an example of floating point code that loads a variable, multiplies it with a constant, and stores the result:

```
                  fld qword ptr x
 x := x * 2.3;    fmul qword ptr flt2_3
                  fstp qword ptr x
```

Formally, we now assume that the program P to be analyzed actually executes two types of assignment statements: (1) *floating-point (FP) assignments* of the form $v := e$ or $[e'] := e$ where v is a floating-point program variable (register) or e is an expression in some larger, possibly undecidable, theory $\mathcal{T}' \supset \mathcal{T}$ including the decidable theory $\mathcal{T}$; and (2) regular (i.e., non-FP) assignment statements as defined in the previous section. For simplicity, we will use the terms *FP assignments* and *non-FP assignments* to distinguish both assignment types in what follows.

For a 32-bit processor, x86 memory addresses $[e']$ are defined by expressions e' of the form "base+index*scale+offset", where base and index are general-purpose 32-bit registers, scale is the constant 1, 2, 4, or 8, and offset is a 32-bit constant value. This convention holds regardless of the type of the value being stored at the address (e.g., a single byte, a 32-bit value, or a floating-point value).

After executing FP assignments, a subsequent conditional statement `if` e `then` C' `else` C'' may have its expression e be dependent on an FP value. We then call it an *FP-dependent conditional statement*.

For instance, the conditional statement in the code below is FP-dependent:

```
double x; int c;            ...
                          fld qword ptr x
    ...                   fcomp flt2_3
if (x > 2.3) {            fnstsw ax
  c: = 1;                 and ah, 41h
}                         jnz l
    ...                   mov dword ptr c, 1
                        l:
                            ...
```

Note that there are no branching FP instructions in x86. There are different idioms for implementing FP-dependent conditional structures in assembly language, one of which is illustrated in the above example. All idioms have in common that the EFLAGS register is set according to the result of an FP computation. This is required since all x86 conditional jumps (and moves) depend on EFLAGS.

Still, extending symbolic execution to handle all or most FP instructions would require adding hundreds of cases to Figure 1, as well as a constraint solver to solve those constraints, which would generate many new tests with new FP input values and would make path explosion even worse.

Our goal is to obtain a theorem similar to Theorem 2 to prove memory safety of executions involving FP computations, even though we cannot precisely symbolically reason about the larger FP theory $\mathcal{T}'$. How can we achieve this?

4. Dealing with FP Instructions

We now describe an approach to proving memory-safety of FP computations *without* expanding symbolic execution, constraint generation, and constraint solving to theory $\mathcal{T}'$. Our approach combines a local, lightweight, path-insensitive, "may" static analysis of FP instructions with a global, high-precision, context and path-sensitive, "must" dynamic analysis of non-FP instructions. The aim of this combination is (1) to prove the memory-safety of each individual FP instruction in their specific run-time execution context, and (2) to prove a form of non-interference between the FP-part and the non-FP part of the program in order to prove memory safety of indirect FP-dependencies through conditional statements and casts from FP-values to non-FP values. In this section, we present a simple algorithm following this general approach, while the next section presents a refined algorithm.

```
* Procedure checkAssignment($C$) =
*   if $C$ is an FP−assignment  // new case
*     $\forall m \in WriteAddrOrReg(C) : S = S + [m \mapsto$FP−tag]
*   return

Procedure executeSymbolicFP($P$,$I$) =
  initialize $M_0$ and $S_0$
  path constraint $pc$ = true
  $C$ = getNextCommand()
  while ($C \neq$ stop)
    match ($C$):
     case ($v := e$):
        $M = M + [\&v \mapsto$evalConcrete($e$)]
        $S = S + [\&v \mapsto$evalSymbolicFP($e$)]
*       checkAssignment($C$)
      case ([$e_1$] := $e_2$):
        $M = M +$ [evalConcrete($e_1$) $\mapsto$evalConcrete($e_2$)]
        evalSymbolicFP([$e_1$]) // passive and active check
        $S = S +$ [evalConcrete($e_1$) $\mapsto$evalSymbolicFP($e_2$)]
*       checkAssignment($C$)
      case (if $e$ then $C'$ else $C''$):
        $b$ =evalConcrete($e$)
        $c$ =evalSymbolicFP($e$)
*       if $c$ = FP−tag
*           error('FP−tag−dependent test detected')
        if $b$ then $pc = pc \wedge c$
        else $pc = pc \wedge \neg c$
    $C$ = getNextCommand() // end of while loop
```

Figure 3. New symbolic execution extended to FP assignments.

The first algorithm starts by computing *statically* for each FP-assignment command C a set $WriteAddrOrReg(C)$ of *memory addresses or regular (non-FP) registers* that are being written to during the execution of C.

Example 1. For the FP assignment C

```
fld [esi+eax]
```

which loads the FP value stored at address `esi+eax` into the FP stack, $WriteAddrOrReg(C) = \emptyset$. □

Example 2. For the FP assignment C

```
fstp qword ptr [edi+ecx]
```

which loads the FP value stored on top of the FP stack into memory at address `edi+ecx`, $WriteAddrOrReg(C) = \{(\texttt{edi+ecx}) \ldots (\texttt{edi+ecx+7})\}$ (where $(\texttt{e})$ denotes the address obtained by evaluating the expression `e` and $[x \ldots y]$ denotes an interval from address x to address y). □

Example 3. For the FP assignment C

```
movd eax, xmm0
```

which loads part of the FP value stored in the FP register `xmm0` into the non-FP register `eax`, $WriteAddrOrReg(C) = \{\texttt{eax}\}$. □

The sets $WriteAddrOrReg(C)$ can be computed for each FP-assignment C statically at compile time or on-demand at run time. These sets are much easier to define and compute than a precise symbolic execution of C. In fact, they provide a *conservative approximation* of the side-effects of C during symbolic execution as shown in Figure 3.

```
1    evalSymbolicFP($e$) =
2      match ($e$):
3        case $v$: // Program variable $v$
4          return $S(\&v)$
5        case +($e_1$ , $e_2$): // Addition
6          $f_1$ = evalSymbolicFP($e_1$)
7          $f_2$ = evalSymbolicFP($e_2$)
8 *        if $f_1$ =FP−tag or $f_2$ =FP−tag
9 *            return FP−tag
10         if $f_1$ and $f_2$ are constants
11             return evalConcrete($e$)
12         else
13             return createExpression('+', $f_1$, $f_2$)
14       case [$e_1$]: // Address dereference
15         $f_1$ = evalSymbolicFP($e_1$)
16 *       if $f_1$ =FP−tag
17 *           error('FP−tag−dependent address detected')
18         $c$ =evalConcrete($e_1$)
19         $mr_c$ =getMemoryRegion($c$)
20         if $mr_c$ is undefined // passive check
21             error('memory access violation at address $c$')
22         if $f_1$ is not a constant
23             $pc = pc \wedge (0 \leq (f_1 - mr_c.base) \leq mr_c.size)$ // active check
24         return $S(c)$
25         etc.
26 *     default: // any FP−specific expression
27 *       return FP−tag
```

Figure 4. New symbolic expression evaluation with FP-tag.

Figure 3 is similar to Figure 2 *except for the lines prefixed by **. Whenever an assignment statement C is encountered during symbolic execution, the function `checkAssignment()` is called (in lines 15 or 20) to check whether C is an FP-assignment. If so, regular symbolic execution of the assignment C is replaced[3] as follows: the symbolic value of every memory address and register in *WriteAddrOrReg*(C) becomes the special symbolic value *FP-tag* in the symbolic store (line 3).

FP-tag is a single symbolic value that represents *all* possible floating-point (concrete) values during symbolic execution, as well as all symbolic expressions that depend on an FP value, hence including all possible symbolic expressions in theory $\mathcal{T}' \setminus \mathcal{T}$. Symbolic expression evaluation is easily extended to handle FP-tags as illustrated in Figure 4. The new value FP-tag is an "absorbing element" with respect to symbolic expression evaluation: the value of any expression containing FP-tag is FP-tag (e.g., see lines 8-9 of Figure 4). Also, any FP-specific expression (i.e., in $\mathcal{T}' \setminus \mathcal{T}$) also returns FP-tag (lines 26-27 of Figure 4).

As in regular symbolic execution, all memory address dereferences are checked both passively and actively for memory access violations (see Section 2). If any memory address ever depends on an FP-tag, an error is (conservatively) generated in line 17 of Figure 4.

The algorithm of Figure 3 also generates an error if the Boolean expression of any conditional statement ever becomes the value FP-tag (lines 24-25).

Let $M(m, s)$ and $S(m, s)$ denote the value of memory location m in state s in the concrete store M or symbolic store S, respectively. Given any two states s and s', we write $s \equiv_{FP} s'$ if $\forall m : M(m, s) = M(m, s')$ or $S(m, s) = S(m, s') =$ FP-tag. Thus, the $\equiv_{FP}$ equivalence is very tight: two states that are $\equiv_{FP}$ equivalent can only differ in any memory location by the concrete floating-point value abstracted by the symbolic value FP-tag.

We extend the notation to define equivalence classes of program executions: given two program executions $w = s_0 \xrightarrow{C_1} s_1 \ldots \xrightarrow{C_n} s_n$ and $w' = s'_0 \xrightarrow{C'_1} s'_1 \ldots \xrightarrow{C'_n} s'_n$, we write $w \equiv_{FP} w'$ if $\forall i : s_i \equiv_{FP} s'_i \wedge C_i = C'_i$.

Theorem 3. *(memory safety) Given a program P as defined above with FP and non-FP assignments, if the algorithm of Figures 3 and 4 does not generate any error for an execution w, then for all executions w′ such that $w' \equiv_{FP} w$, we have $w' \equiv_{P+B} w$, and all executions w′ are memory safe.*

Proof: We first show that if the algorithm does not generate any error for execution w, then $w \equiv_{FP} w'$ implies $w \equiv_{P+B} w'$. First, during the execution w, no conditional statement ever depends on an FP-tag (otherwise contradiction by lines 24-25 of Figure 3), therefore $w \equiv_P w'$. Second, all memory accesses during the execution w are checked in lines 14-24 of Figure 4 (due to calls to `evalSymbolicFP()` in lines 14, 18, 19 and 23 of Figure 3); moreover, no memory address ever depends on an FP-tag (otherwise an error would be generated in line 17 of `evalSymbolicFP()`). This implies $w \equiv_{P+B} w'$.

Since w is memory safe (any non-memory safe access would trigger an error by line 21 of `evalSymbolicFP()`), $w \equiv_{P+B} w'$ implies that w' is memory safe, too. □

Corollary 4. *If the number of program executions w that are distinct with respect to $\equiv_{P+B}$ is finite, all those can be enumerated by a directed search and, if the algorithm of Figures 3 and 4 never generates any error for any of these executions, the entire program P is memory safe.*

Note that if the algorithm of Figures 3 and 4 reports an error of the type "FP-tag-dependent address or test detected", this does not necessarily imply that the program is not memory safe. Indeed, the injection of FP-tags in line 3 of `checkAssignment()` and their propagation using rules like in lines 8-9 of `evalSymbolicFP()` are *conservative*: the FP-tag value abstracts and blends together concrete values that might have been otherwise distinguished by FP constraints injected in the path constraint either in lines 26-27 of `executeSymbolicFP()` or in line 23 of `evalSymbolicFP()`, *if* symbolic execution and constraint solving had been extended to handle FP constraints in $\mathcal{T}'$.

5. Dealing with FP-dependent Conditional Statements

The previous algorithm reports an error whenever the Boolean expression e of any conditional statement depends on an FP-tag. Unfortunately, FP-dependent conditional statements that test floating-point values are frequent in practice, as illustrated in the second example in Section 3 and as will be discussed later in the experiments Section 7. In those cases, the previous algorithm is not sufficient, as it will generate too many alarms.

In this section, we present a refined algorithm that can eliminate many of those alarms. The main idea is to treat an entire if-then-else block depending on an FP-tag as a single FP instruction: a lightweight static analysis is used to over-approximate all possible executions inside that block, starting from the conditional statement C until its unique immediate postdominator instruction *ipdom*(C), and to com-

[3] To keep the notation simple, in case of FP assignments, we assume the symbolic store updates in lines 14 and 19 of Figure 3 are no-ops (e.g., when v is a FP variable) that are subsumed by the updates in line 3 of `checkAssignment()`.

```
1  Procedure executeSymbolicFPV2($P$,$I$) =
2    initialize $M_0$ and $S_0$
3    path constraint $pc =$ true
4    $C$ = getNextCommand()
5    while ($C \neq$ stop)
6      match ($C$):
7       case ($v := e$):
8          $M = M + [\&v \mapsto$evalConcrete$(e)]$
9          $S = S + [\&v \mapsto$evalSymbolicFP$(e)]$
10         checkAssignment($C$)
11       case ($[e_1] := e_2$):
12         $M = M + [$evalConcrete$(e_1) \mapsto$evalConcrete$(e_2)]$
13         evalSymbolicFP($[e_1]$) // passive and active check
14         $S = S + [$evalConcrete$(e_1) \mapsto$evalSymbolicFP$(e_2)]$
15         checkAssignment($C$)
16       case (if $e$ then $C'$ else $C''$):
17         $b =$evalConcrete$(e)$
18         $c =$evalSymbolicFP$(e)$
19 *       if $c =$ FP−tag
20 *           if ($AddrReg(C) = unsafe$) or
21 *              ($\exists reg \in AddrReg(C) : S(\&reg) \neq M(\&reg)$)
22 *                  error('Unsafe FP−dependent conditional')
23 *           $\bar{C} = C$
24 *           while ($\bar{C} \neq ipdom(C)$) $\bar{C}$ = getNextCommand()
25 *           $\forall m \in WriteAddrOrReg(C) : S = S + [m \mapsto$FP−tag$]$
26 *           $C = \bar{C}$; continue // end of while loop
27         else if $b$ then $pc = pc \wedge c$
28              else $pc = pc \wedge \neg c$
29     $C$ = getNextCommand() // end of while loop
```

Figure 5. New symbolic execution extended to FP assignments and FP-dependent conditionals.

pute two sets $AddrReg(C)$ and $WriteAddrOrReg(C)$ for the entire block starting at C:

- $AddrReg(C)$ is a set of regular (non-FP) *registers* whose value at the time of executing C may be used to compute a memory address being read or written during the execution of C to $ipdom(C)$; if any value from memory may be used to compute an address (multiple dereference) during the execution of C to $ipdom(C)$, $AddrReg(C)$ is set to a special value "unsafe".
- $WriteAddrOrReg(C)$ is a set of *memory addresses or regular (non-FP) registers* that may be written to during the execution of C to $ipdom(C)$; if a memory address cannot be expressed by the static analysis for the time immediately before executing $ipdom(C)$ (e.g., a pointer register is overwritten after an access), $AddrReg(C)$ is set to "unsafe".

$ipdom(C)$ can be efficiently determined using standard algorithms [13]; the two sets $AddrReg(C)$ and $WriteAddrOrReg(C)$ are over-approximated and a default value of "unsafe" is used for $AddrReg(C)$ when complicated code is encountered. There are different ways to cheaply compute this over-approximation. In Section 6 we outline a simple algorithm that yields sufficiently precise approximations without full-blown symbolic execution. Before executing the program, this static analysis algorithm is run to pre-compute $ipdom(C)$, $AddrReg(C)$ and $WriteAddrOrReg(C)$ for every conditional statement C in the program under test.

Figure 5 presents the refined symbolic execution algorithm which is similar to the one of Figure 3 except for the new lines 19-26 prefixed with a *. Now, whenever symbolic execution hits an FP-dependent conditional statement C, instead of immediately reporting an error, the refined algorithm attempts to proceed by "skipping" the entire if-then-else block. First, if any register in the statically pre-computed $AddrReg(C)$ has any symbolic value, i.e., is either input-dependent or FP-tag, an error is generated (lines 20-22). Otherwise, symbolic execution proceeds until $ipdom(C)$ is reached (lines 23-24). Just before resuming symbolic execution at $ipdom(C)$ (line 26), the symbolic value of every memory address and register in $WriteAddrOrReg(C)$ becomes FP-tag in the symbolic store (line 25).

Given two program executions w and w', we write $w \equiv_{P+FP} w'$ if $w \equiv_{FP} w'$ *except* at FP-dependent conditional statements C where they are allowed to take different branches and execute different commands *until either* both of them eventually reach $ipdom(C)$, *or* at least one of them never reaches $ipdom(C)$ (due to early termination or non-termination). Thus, the equivalence relation $\equiv_{P+FP}$ is weaker than $\equiv_P$ and $\equiv_{FP}$. We also write $w \equiv_{P+FP+B} w'$ if $w \equiv_{P'+FP} w'$ where P' is P extended with bound checks for all memory accesses.

A program execution w is called *attacker memory safe* if every buffer access during w in the program P extended with bound checks for all memory accesses is *either* within bounds (i.e., is memory safe) *or* is *input-independent*, i.e., its address has no input-dependent symbolic value, and hence is not directly controllable by an attacker through the untrusted input interface.

Thus, the notion of "attacker memory safe" is weaker than memory safety: if a program execution is memory safe, this implies that the execution is attacker memory safe, while the converse does not hold. Nevertheless, in the context of security testing, we are primarily interested in attacker memory safety since buffer overflows where the overflow is fixed and cannot be directly controlled by the attacker are likely much harder to exploit. We now show that the algorithm of Figure 5 can only guarantee this weaker form of memory safety.

Theorem 5. *(attacker memory safety) Given a program P as defined above with FP and non-FP assignments and regular and FP-dependent conditional statements, if the algorithm of Figures 5 and 4 does not generate any error for an execution w, then for all executions w' such that $w' \equiv_{FP} w$, we have $w' \equiv_{P+FP+B} w$, and all executions w' are attacker memory safe.*

Proof: We show that if the algorithm does not generate any errors for execution w, then $w \equiv_{FP} w'$ implies $w \equiv_{P+FP+B} w'$. Since $w \equiv_{FP} w'$, for each conditional statement C reached by those executions, either the conditional statement does not depend on an FP-tag and the two executions execute the same branch at C and subsequent instructions, or the conditional statement depends on an FP-tag and the two executions are allowed to diverge until $ipdom(C)$ is reached. Moreover, all possible side-effects of all possible executions between C and $ipdom(C)$ are covered with FP-tags when $ipdom(C)$ is executed (line 25 of `executeSymbolicFPV2()`). Thus, if s and s' denote the states of w and w' when they both reach $ipdom(C)$, respectively, we have $s \equiv_{FP} s'$. This implies that $w \equiv_{P+FP} w'$.

During the execution of w, all memory accesses performed during symbolic execution are checked to be memory safe in lines 14-24 of `evalSymbolicFP()` (called in lines 9, 13, 14 and 18 of `executeSymbolicFPV2()`). Moreover, *all* memory accesses that can possibly happen during *all* possible program executions between every FP-conditional statement C and $ipdom(C)$ are conservatively represented by $AddrReg(C)$, and all those memory accesses are checked in lines 20-21 to be *both* input- (i.e., attacker-) independent *and* FP-tag-independent (otherwise an error would be generated in line 22). Therefore, no memory access ever depends on an FP-tag and we have $w \equiv_{P+FP+B} w'$.

Furthermore, all memory accesses in w are either memory safe during symbolic execution or input-independent between FP-dependent conditional statements C and their *ipdom*(C). By definition, w is then attacker memory safe. Since $w \equiv_{P+FP+B} w'$, w' is attacker memory safe too. □

The correctness guarantees provided by Theorem 5 are weaker than those provided by Theorem 3, since attacker memory safety is weaker than memory safety. Indeed, executions w' such that $w \equiv_{FP} w'$ can take different executions paths between FP-dependent conditional statements C and their *ipdom*(C), and some of those other paths may trigger memory access violations or other runtime errors (such as division-by-zero or infinite loops). Since we do not generate FP constraints for those conditional statements C, we cannot generate tests to exercise those paths and hit those errors. Nevertheless, our algorithm can guarantee that no such path ever contains an input or FP-dependent memory access.

6. Prototype Implementation

We have implemented our approach as an extension to the existing whitebox fuzz testing tool SAGE [10]. For exploring execution paths of a program under test, the tool starts from a well-formed input file and systematically manipulates input bytes to drive execution into new branches.

We extended the symbolic execution engine of SAGE to incorporate our new algorithm in Figure 5. To calculate the *AddrReg* and *WriteAddrOrReg* sets, we built a separate static analysis tool that processes the binary under test and all referenced dynamic libraries. The static analysis tool is based on Microsoft Vulcan [4], a mature framework for parsing, analyzing, and instrumenting compiled binaries. Dynamic libraries (DLLs) can be conditionally loaded at runtime, and it is not always possible to statically determine the full set of DLLs that may be accessed. If symbolic execution encounters an instruction belonging to the address space of a library for that no static information has been generated, the static analysis has to be invoked on that particular library to generate the missing information.

Static Information. For an executable or DLL, our tool computes the *AddrReg* and *WriteAddrOrReg* sets for all individual FP instructions and for all conditional jumps. We tried to keep the static analysis as fast and simple as possible but as precise as necessary to prove FP dependent conditional jumps to be attacker memory safe (provided that the values of registers in *AddrReg* are input-independent and not FP-tag). Without knowledge about the actual semantics of an instruction, disassemblers such as the one included in Vulcan are still able to extract source and destination operands from it by decoding the relevant bytes in the instruction stream. Typically, elaborate tables are used for mapping bytes to instructions, which also expose implicit register operands (e.g., `eax` is an implicit source and target in `mul cx`). When processing an individual FP instruction, target operands are added to the *WriteAddrOrReg* set.

The *WriteAddrOrReg* and *AddrReg* sets for conditionals need to capture the effects of all control-dependent instructions. We have implemented a flow insensitive interprocedural analysis for determining both sets. Both branches of each conditional jump are explored up to but not including the immediate postdominator of the jump. For each instruction in the branches, its destination operands are added to *WriteAddrOrReg*, and the base and index registers of any memory operands are added to *AddrReg*. If the intersection of both sets is non-empty, i.e., if any of the dereferenced registers is manipulated in the control-dependent block, our analysis reports the conditional as unsafe. For example, consider the following loop controlled by an FP expression:

```
input double x;
int a[10]; int i=0;
while (x > 2.3)
   x = x/2.4;
   a[i++] = (int)x;
```

Our static analysis would return "unsafe" for the condition controlling the loop when analyzing it up to its immediate postdominator (which is the next statement after the while loop). This is necessary, since pointer manipulation in an FP-dependent control block could be vulnerable to an attack that might not be detected by the dynamic analysis .

When all pointers are constant within a conditional block, all target memory addresses can be easily determined and represented in a single pass. No special treatment of loops is required, since the analysis is flow and path insensitive. In our implementation, memory locations in *WriteAddrOrReg* can be of the form [*base* + *offset*], where *base* is a register and *offset* an integer. The restriction that pointers remain constant also excludes multiple dereference within a single block, which in x86 requires assigning an address to an intermediate register.

Disallowing any changes to dereferenced registers is too strict for handling the stack pointer (`esp`) and push/pop. E.g., parameters of function calls are pushed onto the stack in Windows standard calling convention, thereby modifying and dereferencing the stack pointer in one instruction.

We use a standard path-insensitive stack-height analysis supplied by Vulcan to determine at every instruction the offset of `esp` from the base of the procedure's stack frame. In compiler generated code, this offset is always constant for a certain instruction. We can thus replace references to `esp`-based local variables with equivalent stack frame based memory locations in the *WriteAddrOrReg* set. At the immediate postdominator of the conditional jump, these locations are then translated back to `esp` based addresses.

Function calls within in the control-dependent block are allowed. Our tool recursively calculates *AddrReg* and *WriteAddrOrReg* for complete procedures in standard bottom up fashion and inserts them at the call site. Care must be taken to correctly translate stack frame relative addresses in the *WriteAddrOrReg* set of the callee into the stack frame of the caller. Some procedures set up and use a frame pointer in `ebp` for referencing local variables, so `ebp` would show up in both sets. We solve this problem by using the same translation as for `esp`, and express all `ebp` based local variables as stack frame relative addresses (in Visual C, `ebp` always has an offset of 4 from the stack frame).

This approach also handles recursive or mutually recursive functions, since it is not necessary for *WriteAddrOrReg* to record updates to stack frames which have become invalid at the immediate postdominator of the original conditional statement. Any other side effects are over-approximated by one pass of a function, since other pointers than the stack and frame pointer have to remain constant.

Control Flow Information. The calculation of immediate postdominators for conditional jumps requires precise control flow information. Our prototype correctly handles procedures with multiple exits, non-returning procedure calls (e.g., `ExitProcess`), inter-procedure jumps (e.g., from shared error handling code), and the tail-jump optimization; it currently does not support implicit control flow through exceptions raised by instructions or in callees. If an immediate postdominator cannot be statically determined for a conditional jump (e.g., because one branch terminates), the implementation reports the conditional to be unsafe.

We rely on Vulcan to decode instructions, identify procedures, and provide control flow information. Soundness of our implementation therefore depends on the soundness of Vulcan's analysis.

SSA optimizations in the C runtime library. With our assumption that floating point and SSE instructions are used exclusively for payload processing, we did not expect pointers to be modified by them. However, it turned out that the implementations of memset and memcpy in Microsoft's C runtime use SSA instructions for speed, if the CPU supports them and if the memory blocks happen to be 16-byte aligned.

Both cases lead to memory locations falsely being tagged as floating point data (set to FP-tag), including pointers inside larger structures. For memset, we extended the static analysis to understand the PXOR (16-byte XOR) and MOVDQA (16-byte move) instructions and to perform a simple intraprocedural constant propagation. This suffices to identify the SSA move instructions inside memset as assignments of a constant integer value, for which *WriteAddrOrReg* is empty. For memcpy, the symbolic execution should be aware that memory areas are being copied, since whenever a portion of input-dependent data is copied, its symbolic value should be preserved. We handle this as a special case during symbolic execution of the specific code sequence of memcpy to keep track of the size, source and target addresses of the memory to be copied.

7. Experimental Results

We report in this section detailed results of experiments with our implementation and test drivers that invoke the built-in parsers for JPEG, GIF, and ANI image files embedded in Windows Vista. Those parsers are implemented in code spread across various Windows DLLs. For each parser, we first preprocessed all required libraries using our static analysis tool. Many of the libraries are unrelated to the actual parsing but are used for purposes like file access or setting up the Microsoft COM services. This foundation of common library code is shared by all parsers, so there is a rather large overlap in the static information used for them.

Static Preprocessing. Table 1 gives a breakdown of the results for the static analysis phase, both individually for each DLL and as a total for each parser. We analyzed 20 DLLs in total; the JPEG parser loaded 16, the one for GIF 19, and the one for ANI 15 DLLs. Processing all required DLLs per parser took about 5 to 10 min on a regular desktop machine. One of the DLLs, shell32.dll, unfortunately was protected by binary obfuscation and could not be processed by our prototype.

Our tool created the static information for *all* conditional jumps in the program, not only for those which are FP dependent at runtime, since it is not generally possible to statically determine the set of FP-dependent conditionals. For over 80% of the conditional jumps, our lightweight static analysis is not precise enough to prove anything, and has to flag the conditional as *unsafe*. However, we do not try to statically prove memory safety of arbitrary code, but are only interested in those conditional statements which become FP-dependent at runtime. All other conditionals should eventually be explored by the algorithm for directed testing and are thus already covered. We designed the static analysis towards the patterns we saw for conditional jumps inside floating point program logic, extending its power as required to cover all or almost all cases. In particular, our static analysis reports as unsafe a conditional statement if in one of the branches the same pointer is both modified and dereferenced (with the exception of the stack and frame pointers, as explained in the previous section). This includes multiple pointer dereferences (e.g., expressions such as **p), loops over arrays, and long code sequences where the same register is used for different pointers. FP-dependent conditionals in the file parsers we looked at are usually relatively short and/or limit their effects on FP registers, so we were able to avoid implementing a more expensive analysis. We did, however, have to make our analysis interprocedural, as some of the FP-dependent conditionals contained function calls.

About 6% of all conditional jumps are determined to be unconditionally memory *safe*, which is the case if the conditional block contains no dereferences in either branch and *AddrReg* is empty. The remaining conditionals do contain dereferences and are *conditionally safe*: their *AddrReg* is nonempty and the registers it contains need to be checked for FP-dependent values during symbolic execution.

Symbolic Execution. In designing our experiments, we were interested in checking whether the missing floating point support could have caused our existing whitebox fuzzer to miss bugs. If our assumption that FP instructions do not interfere with security critical code is correct, ignoring FP instructions did not miss any security bugs. We therefore focused on replaying the symbolic execution of interesting input files to determine whether all executions terminate without issuing warnings, which could be raised due to unsafe FP-dependent conditionals or FP-dependent memory accesses.

We ran SAGE's symbolic execution on diverse seed files and disabled constraint solving, so that for each seed file we observed one complete symbolic execution trace. We used twelve different seed files per format, randomly selected from a suite of regression tests. The files were of various sizes, to a combined total size of 238 kB in the case of JPEG. Only for JPEG we saw a significant amount of new instructions being covered compared to a single seed file, and in fact some of the seed files caused more DLLs to be loaded by the parser. Still, no new warnings were raised compared to the symbolic execution of the trace for a single seed file per format. Overall, our implementation of symbolic execution with FP tags and processing of static information consistently implies a 20% runtime overhead compared to symbolic execution without FP tags, and therefore comes at an overall marginal runtime cost.

Table 2 lists the results obtained from symbolically executing the test drivers on one seed file each; as mentioned above, results for the other input files are very similar. The total runtime for the extended symbolic execution were 101 secs for JPEG, 73 secs for GIF, and 5 seconds for ANI. These executions were performed with small seed files of 1,092 bytes for JPEG, 2,957 bytes for GIF, and 2,512 bytes for ANI. Numbers in Table 2 are split between *occurrences* of instructions (including repeated executions) and *unique* instructions, and also between instructions in the *full* trace from initialization to termination and instructions executed after at least one *input* byte has been read in the execution trace.

For instance, we ran the JPEG test driver and traced 26.7 million instructions (86763 unique), of which 22 million occurred after reading input. It executed only 89 unique FP instructions after the input was read, a large portion of which was part of an inner loop of optimized SSE2 instructions to

DLL	JPEG	GIF	ANI	All instr.	FP instr.	Conditionals	Safe	Cond. Safe	Unsafe	Time
advapi32	✓	✓	✓	156442	75	13370	3.6%	10.0%	86.3%	27s
clbcatq	✓	✓		114240	100	12668	24.5%	7.8%	67.7%	27s
comctl32		✓	✓	376620	344	31335	6.1%	14.6%	79.3%	47s
gdi32	✓	✓	✓	81834	366	8785	3.6%	9.9%	86.5%	11s
GdiPlus		✓		476642	32147	42154	5.4%	13.4%	81.3%	184s
imm32	✓	✓	✓	26178	0	2712	5.3%	4.9%	89.9%	6s
kernel32	✓	✓	✓	15958	12	15958	4.1%	12.4%	83.4%	33s
lpk	✓	✓	✓	5389	45	658	8.1%	16.1%	75.8%	2s
msctf	✓	✓	✓	159228	357	13985	4.5%	8.3%	87.1%	31s
msvcrt	✓	✓	✓	147640	5757	16260	6.0%	12.1%	81.9%	35s
ntdll	✓	✓	✓	207815	649	18876	5.1%	12.6%	82.3%	40s
ole32	✓	✓		367226	99	32677	4.8%	7.1%	88.1%	81s
oleaut32	✓	✓		148777	1335	15484	7.1%	9.6%	83.3%	25s
rpcrt4	✓	✓	✓	240231	57	18603	5.6%	9.5%	84.9%	31s
shell32		✓	✓	-	-	-	-	-	-	-
shlwapi		✓	✓	73092	0	6914	7.5%	12.3%	80.3%	9s
user32	✓	✓	✓	121223	0	11314	7.3%	12.6%	80.0%	16s
usp10	✓	✓	✓	79990	2	8394	7.7%	11.9%	80.5%	11s
uxtheme	✓	✓	✓	62276	110	5488	5.9%	10.9%	83.2%	7s
WindowsCodecs	✓			193415	6370	16926	4.4%	7.4%	88.2%	35s
JPEG (Total)				2127862	15334	212158	6.4%	9.8%	83.8%	418s
GIF (Total)				2860801	41455	275635	6.4%	11.1%	82.5%	623s
ANI (Total)				1753916	7774	172652	5.5%	11.7%	82.8%	306s

Table 1. Results from static analysis of DLLs used by the parsers.

		All instructions		FP instructions		Total FP cond.		Safe FP cond.		Unsafe FP cond.	
		Full	Input	Full	Input	Full	Input	Full	Input	Full	Input
JPEG	Occurrences	26712705	21983468	7826	7320	45	4	39 (87%)	4	6 (13%)	0
	Unique	86763		104	89	28	1	26 (93%)	1	2 (7%)	0
GIF	Occurrences	8952406	4786801	3856	0	435	0	299 (69%)	0	136 (31%)	0
	Unique	133958		68	0	36	0	32 (89%)	0	4 (11%)	0
ANI	Occurrences	1581268	1207886	134	39	41	21	35 (85%)	15	6 (15%)	6
	Unique	29722		16	13	27	7	25 (93%)	5	2 (7%)	2

Table 2. Results from FP-extended symbolic execution for one input file per parser.

perform a discrete cosine transform. All three parsers executed FP instructions at some point; in the case of GIF, however, all FP instructions occurred before any input was read. None of the FP instructions and also none of the regular non-branch instructions was found to be unsafe, i.e., *no instruction ever dereferenced an FP-dependent value.*

All three parsers executed conditional jumps where the EFLAGS register was tagged as being FP-dependent, thus confirming the need for the over-approximating static analysis we introduced in Section 5. Table 2 shows the total number of such FP-dependent conditional jumps, as well as the number of safe and unsafe FP conditionals. An FP-dependent conditional C is listed as *safe* if the block up to *ipdom*(C) was successfully over-approximated using C's static information and was found to be memory safe. It is listed as *unsafe* if static analysis has failed and *AddrReg* in its static information was set to unsafe, or if *AddrReg* contains registers holding FP-tagged values at the time of the jump (we never encountered the latter case in our experiments). The JPEG trace contained 45 occurrences of FP-dependent conditional jumps, of which 6 (corresponding to 2 unique jumps) had an unsafe precomputed *AddrReg* and 39 were found to be safe. However, all unsafe conditionals were executed before any data was read from the input file; if the attacker is only able to control the input file, which corresponds to our threat model, it is not possible for him to control the branching behavior of these instructions. Hence, a bug in these conditionals would surface for all or no inputs, and the entire execution is attacker memory safe.

In the GIF trace 4 out of 36 unique FP-dependent conditionals raised warnings, but similar to JPEG, all of them occurred before any input was read.

For ANI, however, we observed two unsafe unique conditional jumps after reading the input file. These were the same two conditional jumps from the same DLL (uxtheme.dll) as in the JPEG trace. Both jumps had an "unsafe" value for *AddrReg*; the static analysis was unable to deduce memory safety because in both cases one of the branches called a rounding function containing sophisticated error handling code that involves multiple functions for notifying an attached debugger and setting a global error status. However, after a careful visual inspection of the disassembled code, we are confident the function containing both jumps belongs to initialization code common with the JPEG parser, the only difference being that this initialization code is invoked later in the ANI case, after some inputs have already been read from the input files. Regular runtime symbolic execution also indicates that no regular input ever flows in the entire DLL containing this initialization code, although it cannot prove that no regular inputs are ever being cast into some untracked FP-value that later influences the execution of this initialization code. We nevertheless believe that it is unlikely that this code is ever called with attacker controllable inputs. A more sophisticated static and/or dynamic analysis would be needed to prove this automatically.

Although our static analysis declares 80% of all (FP-dependent and FP-independent) conditional jumps as being unsafe, it is good enough to reduce the number of runtime warnings about unsafe FP-dependent conditional statements to zero or two in the case of ANI. For the benchmarks considered, the algorithms of section 4 and 5 meet our initial goal of designing an ’as lightweight as possible’ static analysis to prove non-interference between floating point code and security critical computations.

Preliminary experiments with other Windows media parsers confirm the key observation made in this paper: for most programs in the application domain we consider, interactions between floating-point computations and memory allocation and indexing are extremely rare. For instance, parsing a sample small WMV (Windows Media Video) file with 27,401 bytes executes 252,715,214 instructions (including 247,256,201 after the first input byte is read), but executes only 499 FP-instructions (i.e., less than 0.001% of the total number of instructions) and 29 FP-tainted conditional statements, which are proved attacker memory safe with our extended symbolic execution in 350 secs. We plan to optimize the memory usage of our current prototype implementation in order to handle longer execution traces and hence larger files and more complicated parsers.

8. Related Work and Discussion

There are many other algorithms combining static and dynamic program analysis, some also aimed at proving memory safety [16] or type checking [1, 6]. Our work can be viewed as following the same general strategy of “prove statically as much as possible, and use runtime checks as a fallback”. However, a perhaps unique feature of our analysis is that the abstract domains used for the static and dynamic parts are quite distinct: we use a simple path-insensitive static analysis targeting FP instructions and tracking memory usage of individual if-then-else blocks, while we use a (bit-)precise runtime symbolic execution to reason about the non-FP part of the program. The key novelty of our approach is the interplay between these two analyses and abstract domains, and the new notion of attacker memory safety that this combination is able to prove. Our analysis also attempts to prove a form of non-interference [18] between FP computations and memory accesses: FP values do not influence memory accesses.

In principle, the general strategy of over-approximating instructions not handled by symbolic execution could be used to prove memory safety in presence of instructions other than FP instructions, or to deal with any theory $\mathcal{T}'$ of constraints outside $\mathcal{T}$ for which we do not want to generate constraints (because $\mathcal{T}'$ is too complex, undecidable, expensive, or a solver is simply not readily available for whatever reason). However, this strategy seems to work well for FP instructions because of the expected non-interference of FP values (payload) with address computations (control). In practice, it is unclear if this strategy would work for other sets of unhandled instructions or constraints.

The static part of our analysis is conservative and handles multi-pointer dereferences (by returning ’unsafe’). However, the dynamic part as presented in Section 2 does not. To handle multiple levels of pointer dereferences and symbolic writes, the memory model used in Section 2 would need to be extended as discussed in [5] for instance. We did not consider this option here to simplify the exposition, as it is an orthogonal issue, and also because multi-pointer dereferences involving untrusted inputs are rare.

Recent work has started to address industrial-strength analysis of FP programs whose functional correctness is critical in some application domains like avionics [15]. Also, there is no doubt that SMT solvers will one day be extended to FP arithmetic. But for proving only memory safety of FP programs in the application domain we considered here, our work shows that precise FP reasoning is often not necessary, which is good news.

Note that our technique is based on the idea that floating point code does not influence (attacker) memory safety. We specifically do not aim to prove correctness of the floating point computations. We also do not consider other application domains (from avionics, spec FP benchmarks, JavaScript programs where variables have floating-point value types by default, etc.) where the non-interference we seek may not hold. It is also trivial to write down toy programs for which our analysis technique would fail (simply cast a double to an integer and then index a buffer with that integer). The goal of our work is specifically to prove attacker memory safety of “nearly-secure” image and video parsers, that are part of Windows and deployed on a billion PCs worldwide. Today, we are not aware of any other practical analysis and tool that can do this for the class and size of applications we consider.

9. Conclusions

In this paper, we introduced a new proof technique for attacker memory safety, which combines lightweight static analysis of floating-point parts of a program with a precise runtime symbolic execution of the rest of the program.

We do not require theorem prover support for floating point, since our goal is not to reason precisely about the floating point logic of the program, but to prove attacker memory safety. Our intuition was verified in the examples we considered: the FP part does not interact in any dangerous way with buffer allocation and indexing.

As future work we plan to extend our combined analysis to cover the remaining cases of FP-dependent conditionals considered unsafe by our prototype. We also plan experiments with more file parsers, including more complex media players. Furthermore, since the combined proof strategy we propose is not generally linked to FP code, we will investigate other areas where dynamic test generation can benefit from replacing precise symbolic execution by a coarse and cheap over-approximation.

References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *Proceedings of POPL’89*, pages 213–227, 1989.

[2] G. Balakrishnan and T. W. Reps. Analyzing memory accesses in x86 executables. In *13th Int’l Conf. Compiler Construction (CC 2004)*, volume 2985 of *LNCS*, pages 5–23. Springer, 2004.

[3] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *ACM CCS*, 2006.

[4] A. Edwards, A. Srivastava, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical Report

MSR-TR-2001-50, Microsoft Research, 2001.

[5] B. Elkarablieh, , P. Godefroid, and M. Levin. Precise Pointer Reasoning for Dynamic Test Generation. In *Proceedings of ISSTA'09*, pages 129–139, Chicago, July 2009.

[6] C. Flanagan. Hybrid type checking. In *Proceedings of POPL'06*, January 2006.

[7] P. Godefroid. Software Model Checking Improving Security of a Billion Computers. In *Proceedings of SPIN'09*, page 1, Grenoble, June 2009.

[8] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of PLDI'2005*, pages 213–223, Chicago, June 2005.

[9] P. Godefroid, M. Levin, and D. Molnar. Active Property Checking. In *Proceedings of EMSOFT'08*, pages 207–216, Atlanta, October 2008. ACM Press.

[10] P. Godefroid, M. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of NDSS'2008 (Network and Distributed Systems Security)*, pages 151–166, San Diego, February 2008.

[11] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2009.

[12] J. Kinder and H. Veith. Jakstab: A static analysis platform for binaries. In *20th Int'l Conf. Computer Aided Verification (CAV 2008)*, volume 5123 of *LNCS*, pages 423–427. Springer, 2008.

[13] T. Lengauer and R. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.

[14] D. Molnar, X. C. Li, and D. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proc. of the 18th Usenix Security Symposium*, Aug 2009.

[15] D. Monniaux. The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(3):1–41, May 2008.

[16] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of POPL'02*, pages 128–139, Portland, January 2002.

[17] S. K. Raman, V. M. Pentkovski, and J. Keshava. Implementing Streaming SIMD Extensions on the Pentium III Processor. *IEEE Micro*, 20(4):47–57, 2000.

[18] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *JSAC*, 21(1):5–19, Jan. 2003.

[19] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of POPL'86*, pages 184–192, St. Petersburgh, January 1986.