Proving Memory Safety of the ANI Windows Image Parser using Compositional Exhaustive Testing

Maria Christakis[1]* and Patrice Godefroid[2]

[1] Department of Computer Science
ETH Zurich, Switzerland
maria.christakis@inf.ethz.ch
[2] Microsoft Research
Redmond, USA
pg@microsoft.com

Abstract. We report in this paper how we proved memory safety of a complex Windows image parser written in low-level C in only three months of work and using only three core techniques, namely (1) symbolic execution at the x86 binary level, (2) exhaustive program path enumeration and testing, and (3) user-guided program decomposition and summarization. We also used a new tool, named MicroX, for executing code fragments in isolation using a custom virtual machine designed for testing purposes. As a result of this work, we are able to prove, for the first time, that a Windows image parser is *memory safe*, i.e., free of any buffer-overflow security vulnerabilities, *modulo* the soundness of our tools and several additional assumptions regarding bounding input-dependent loops, fixing a few buffer-overflow bugs, and excluding some code parts that are not memory safe by design. In the process, we also discovered and fixed several limitations in our tools, and narrowed the gap between systematic testing and verification.

1 Introduction

Systematic dynamic test generation [18, 9] consists of repeatedly running a program both concretely and symbolically. The goal is to collect symbolic constraints on inputs from predicates in branch statements along the execution, and then to infer variants of the previous inputs, using a constraint solver, in order to steer the next execution of the program toward an alternative program path. By systematically repeating this process, the entire set of execution paths of a program can, in principle, be explored. This approach to automatic test generation has become popular over the last several years, and has been implemented in many tools such as EXE [10], jCUTE [33], SAGE [21], Pex [36], KLEE [8], BitBlaze [34], and Apollo [2] to name a few. These tools vary by the programming languages, properties, and application domains they target, but they have all been successful in discovering new bugs missed by more conventional techniques. Notably, SAGE is credited to have found roughly one third

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

of all the security bugs discovered by file fuzzing during the development of Microsoft's Windows 7 [6]. Despite their success and popularity, the tools above have never been used so far for program *verification* of a non-trivial application, i.e., for proving the absence of specific classes of bugs.

In this paper, we show how we used and enhanced these techniques to prove memory safety of the ANI Windows image parser. This parser is responsible for processing structured graphics files to display "ANImated" cursors and icons on more than a billion PCs. Such animated icons are ubiquitous in practice (like the spinning ring or hourglass on Windows), and their domain of use ranges from web pages and blogs, instant messaging and e-mails, to presentations and video clips. The ANI parser consists of thousands of lines of low-level C code spread across hundreds of functions. Yet, this parser is sequential (no concurrency or real-time constraints). It is also of security interest: in 2007, a critical out-of-band security patch was released for code in this parser (MS07-017) costing Microsoft and its users millions of dollars [35, 24]. A motivation for this work was to determine whether the ANI parser is now free of security-critical buffer overflows.

We show how systematic dynamic test generation can be applied and extended to program verification. To achieve this, we address the two main limitations of dynamic test generation, namely imperfect symbolic execution and path explosion. For the former, we extended the tool SAGE to improve its symbolic execution engine so that it could handle all the x86 instructions of that specific ANI parser. To deal with path explosion, we used a combination of function inlining, restricting the bounds of input-dependent loops, and function summarization. We also used a new tool, named MicroX, for executing code fragments in isolation using a custom virtual machine designed for testing purposes. We emphasize that the focus of our work is restricted to proving the absence of attacker-controllable memory-safety violations (as precisely defined in Sect. 3).

At a high-level, the main contributions of this paper are: (1) We report on the first application of systematic dynamic test generation for *verifying* a real, complex, security-critical, entire program. Our work sheds light on the shrinking gap between systematic testing and verification in a model-checking style. (2) To our knowledge, this is the first time that an operating-system (Windows or other) *image parser* has been *proven free of security-critical buffer overflows*. (3) We are also not aware of any past attempts at program verification *without using any static program analysis*; all the techniques and tools used in this work are exclusively dynamic.

This paper is organized as follows. In Sect. 2, we recall basic principles of systematic dynamic test generation and compositional symbolic execution, and briefly present the SAGE and MicroX tools used in this work. In Sect. 3, we precisely define memory safety, show how to verify it compositionally, and discuss how we used and extended SAGE and MicroX for verification. Sect. 4 presents an overview of the ANI Windows image parser. In Sect. 5, we present our verification results in detail. During the course of this work, we discovered several memory-safety violations in the ANI parser code, which are discussed in Sect. 6. We review related work in Sect. 7 and conclude in Sect. 8.

2 Background

2.1 Systematic Dynamic Test Generation

Systematic dynamic test generation [18, 9] consists of repeatedly running a program both concretely and symbolically. The goal is to collect symbolic constraints on inputs from predicates in branch statements along the execution, and then to infer variants of the previous inputs, using a constraint solver, in order to steer the next execution of the program toward an alternative path.

Symbolic execution means executing a program with symbolic rather than concrete values. Assignment statements are represented as functions of their (symbolic) arguments, while conditional statements are expressed as constraints on symbolic values. Side-by-side concrete and symbolic executions are performed using a concrete store M and a symbolic store S, which are mappings from memory addresses (where program variables are stored) to concrete and symbolic values, respectively. For a program path w, a *path constraint* ϕ_w is a logic formula that characterizes the input values for which the program executes along w. Each symbolic variable appearing in ϕ_w is, thus, a program input. Each constraint is expressed in some theory[3] T decided by a constraint solver, i.e., an automated theorem prover that can return a satisfying assignment for all variables appearing in constraints it proves satisfiable.

All program paths can be enumerated by a search algorithm that explores all possible branches at conditional statements. The paths w for which ϕ_w is satisfiable are feasible, and are the only ones that can be executed by the actual program provided the solutions to ϕ_w characterize exactly the inputs that drive the program through w. Assuming that the constraint solver used to check the satisfiability of all formulas ϕ_w is sound and complete, this use of symbolic execution for programs with finitely many paths amounts to program verification.

2.2 Compositional Symbolic Execution

Systematically testing and symbolically executing all feasible program paths does not scale to large programs. Indeed, the number of feasible paths can be exponential in the program size, or even infinite in the presence of loops with an unbounded number of iterations. This *path explosion* can be alleviated by performing symbolic execution *compositionally* [15, 1].

In compositional symbolic execution, a summary ϕ_f for a function (or any program sub-computation) f is defined as a logic formula over constraints expressed in theory T. Summary ϕ_f can be generated by symbolically executing each path of function f, then generating an input precondition and output postcondition for each path, and bundling together all path summaries in a disjunction. Precisely, ϕ_f is defined as a disjunction of formulas ϕ_{w_f} of the form

$$\phi_{w_f} = pre_{w_f} \wedge post_{w_f}$$

where w_f denotes an intraprocedural path in f, pre_{w_f} is a conjunction of constraints on the inputs of f, and $post_{w_f}$ a conjunction of constraints on the outputs of f. An input to a function f is any value that can be read by f, while

[3] A theory is a set of logic formulas.

an output of f is any value written by f. Therefore, ϕ_{w_f} can be computed automatically when symbolically executing the intraprocedural path w_f: pre_{w_f} is the path constraint along path w_f but expressed in terms of the function inputs, while $post_{w_f}$ is a conjunction of constraints, each of the form $v' = S(v)$, where v' is a fresh symbolic variable created for each program variable v modified during the execution of w_f (including the return value), and where $S(v)$ denotes the symbolic value associated with v in the program state reached at the end of w_f. At the end of the execution of w_f, the symbolic store is updated so that each such value $S(v)$ is replaced by v'. When symbolic execution continues after the function returns, such symbolic values v' are treated as inputs to the calling context. Summaries can be re-used across different calling contexts.

For instance, given the function `is_positive` below,

```
int is_positive(int x) {
  if (x > 0) return 1;
  return 0;
}
```

a summary ϕ_f for this function can be

$$\phi_f = (x > 0 \wedge ret = 1) \vee (x \leq 0 \wedge ret = 0)$$

where ret denotes the value returned by the function.

Symbolic variables are associated with function inputs (like x in the example) and function outputs (like ret in the example) in addition to whole-program inputs. In order to generate a new test to cover a new branch b in some function, all the previously known summaries can be used to generate a formula ϕ_P symbolically representing all the paths discovered so far during the search. By construction [15], symbolic variables corresponding to function inputs and outputs are all bound in ϕ_P, and the remaining free variables correspond exclusively to whole-program inputs (since only those can be controlled for test generation).

For instance, for the program `P` below,

```
#define N 100
void P(int s[N]) { // N inputs
  int i, cnt = 0;
  for (i = 0; i < N; i++) cnt = cnt + is_positive(s[i]);
  if (cnt == 3) error(); // (*)
}
```

a formula ϕ_P to generate a test covering the `then` branch `(*)` given the above summary ϕ_f for function `is_positive` can be

$$(ret_0 + ret_1 + \ldots + ret_{N-1} = 3) \wedge$$

$$\bigwedge_{0 \leq i < N} ((s[i] > 0 \wedge ret_i = 1) \vee (s[i] \leq 0 \wedge ret_i = 0))$$

where ret_i denotes the return value of the ith call to function `is_positive`. Even though program `P` has 2^N feasible whole-program paths, compositional test generation can cover symbolically all those paths with at most 4 test inputs: 2 tests to cover both branches in function `is_positive` plus 2 tests to cover both

branches of the `if` statement `(*)`. In this example, compositionality avoids an exponential number of tests and calls to the constraint solver at the cost of using more complex formulas with more disjunctions.

When, where, and how compositionality is worth using in practice is still an open question (e.g., [15, 1, 5, 26]), which we discuss later in this paper.

2.3 SAGE and MicroX

Our ANI verification work was carried out using extensions of two existing tools: SAGE [21] and MicroX [16]. SAGE is a whitebox fuzzer for security testing, which implements systematic dynamic test generation and performs dynamic symbolic execution at the x86 binary level. It is optimized to scale to very large execution traces (billions of x86 instructions) and programs (like Excel). SAGE also implements a limited form of summaries [19] as well as specialized forms of summaries for dealing with floating-point computations [17] and input-dependent loops [22]. The feature for floating-point computations was not used in this work as the ANI parser considered here does not include floating-point instructions, while the latter feature is too limited to deal with all the ANI input-dependent loops—we handled those differently as explained in Sect. 5.2.

MicroX is a newer tool [16] for executing code fragments in isolation, without user-provided test drivers or input data, using a custom virtual machine (VM) designed for testing purposes. Given any user-specified code location in an x86 binary, the MicroX VM starts executing the code at that location, intercepts all memory operations before they occur, allocates memory on-the-fly in order to perform those read/write memory operations, and provides input values according to a customizable *memory policy*, which defines what read memory accesses should be treated as inputs. By default, an input is defined as any value read from an uninitialized function argument, or through a dereference of a previous input (recursively) that is used as an address. This memory policy is typically adequate for testing C functions. No test driver/harness is required: MicroX discovers automatically and dynamically the input/output signature of the code being run. Input values are provided as needed along the execution and can be generated in various ways, e.g., randomly or using some other test-generation tool like SAGE. When used with SAGE, the very first test inputs are generated randomly; then, SAGE symbolically executes the code path taken by the given execution, generates a path constraint for that (concrete) execution, and solves new alternate path constraints that, when satisfiable, generate new input values guiding future executions along new paths.

3 Proving Memory Safety

3.1 Defining Memory Safety

To prove memory safety during systematic dynamic test generation, all memory accesses need to be checked for possible violations. Whenever a memory address a stored in a program variable v (i.e., $a = M(v)$) is accessed during execution, the concrete value a of the address is first checked "passively" to make sure it points to a valid memory region mr_a (as done in standard tools like Purify,

Valgrind and AppVerifier); then, if this address a was obtained by computing an expression e that depends on an input (i.e., $e = S(v)$), the symbolic expression e is also checked "actively" by injecting a new *bounds-checking* constraint

$$0 \leq (e - mr_a.base) < mr_a.size$$

in the path constraint to make sure other input values cannot trigger a buffer overflow or underflow at this point of the program execution [10, 20]. How to keep track of the base address $mr_a.base$ and size $mr_a.size$ of each valid memory region mr_a during the program execution is discussed in work on precise symbolic pointer reasoning [14].

As an example, consider the following function:

```
void buggy(int x) {
  char* buf[10];
  buf[x] = 1;
}
```

If this function is run with `x=1` as input, the concrete execution is memory safe as the memory access `buf[1]` is in bounds. In order to force systematic dynamic test generation to discover that this program is not memory safe, it is mandatory to inject the constraint $0 \leq x < 10$ in the current path constraint when the statement `buf[x]=1` is executed. This constraint is later negated and solved leading to other input values for `x`, such as -1 or 10, with which the function will be re-tested and caught violating memory safety.

A program execution w is called *attacker memory safe* [17] if every memory access during w in program P, which is extended with bound checks for all memory accesses, is either within bounds, i.e., memory safe, or input independent, i.e., its address has no input-dependent symbolic value, and hence, is not controllable by an attacker through the untrusted input interface. A program is called attacker memory safe if all its executions are attacker memory safe.

Thus, the notion of attacker memory safety is weaker than traditional memory safety: a memory-safe program execution is always attacker memory safe, while the converse does not necessarily hold. For instance, an attacker-memory-safe program might perform a flawless and complete validation of all its untrusted inputs, but might still crash (for instance, by accessing the address `NULL`) in error-handling code that is executed exclusively after a trusted system call fails.

Security testing is primarily aimed at checking attacker memory safety since buffer overflows that cannot be controlled by the attacker are not security critical. In the rest of this paper, we focus on attacker memory safety, but we will often refer to it simply as memory safety for convenience.

3.2 Proving Attacker Memory Safety Compositionally

In order to prove memory safety compositionally, bounds-checking constraints need to be recorded inside summaries and evaluated for each calling context.

Consider the following function `bar`:

```
void bar(char* buf, int x) {
  if ((0 <= x) && (x < 10)) buf[x] = 1;
}
```

If we analyze `bar` in isolation without knowing the size of the input buffer `buf`, we cannot determine whether the buffer access `buf[x]` is memory safe. When we summarize function `bar`, we include in the precondition of the function that `bar` accesses the address `buf+x` when the condition $(0 \leq x) \wedge (x < 10)$ holds. A summary for this function executed with, say, `x=3` can then be:

$$(0 \leq x) \wedge (x < 10) \wedge (0 \leq x < mr_{\texttt{buf}}.size) \wedge (\texttt{buf}[\texttt{x}] = 1)$$

Later, when analyzing higher-level functions calling `bar`, these bounds-checking constraints can be checked because the buffer bounds will then be known. For instance, consider the following function `foo` that calls `bar`:

```
void foo(int x) {
  char *buf = malloc(5);
  bar(buf, x);
}
```

If `foo` calls `bar` with `x=3`, the precondition of the above path summary for `bar` is satisfied. The bounds-checking constraint can be simplified with $mr_{\texttt{buf}}.size = 5$ in this calling context and negated to obtain the new path constraint,

$$(0 \leq x) \wedge (x < 10) \wedge \neg(0 \leq x < 5)$$

which after simplification is

$$(0 \leq x) \wedge (x < 10) \wedge ((x < 0) \vee (x \geq 5))$$

This constraint is satisfiable with, say, $x = 7$, and running `foo` and `bar` with that new input value will then detect a memory-safety violation in `bar`.

To sum up, the procedure we use for proving memory safety compositionally is as follows. We record bounds-checking constraints in the preconditions of intraprocedural path-constraint summaries. Whenever a path summary is used in a specific calling context, we check whether its precondition contains any bounds-checking constraint. If so, we check whether the size of the memory region appearing in the bounds-checking constraint is known. If this is the case, we generate a new alternate path constraint defined as the conjunction of the current path constraint and the negation of the bounds-checking constraint, where the size of the memory region is replaced by the current size. We then attempt to solve this alternate path constraint with the constraint solver, which then generates a new test if the constraint is satisfiable.

For real C functions, the logic representations of their pre- and postconditions can quickly become very complex and large. We show later in this paper that, by using summarization sparingly and at well-behaved function interfaces, these representations remain tractable.

We have implemented in SAGE the compositional procedure for proving memory safety described in this section.

3.3 Verification with SAGE and MicroX

In order to use SAGE for *verification*, we *turned on maximum precision* for symbolic execution: all runtime checkers (for buffer overflows and underflows, division by zero, etc.) were turned on as well as precise symbolic pointer reasoning [14], any x86 instruction unhandled by symbolic execution was reported,

every path constraint was checked to be satisfiable before negating constraints, we checked that our constraint solver, the Z3 automated theorem prover [13], never timed out on any constraint, and we also checked the absence of any *divergence*, which occurs whenever a new test generated by SAGE does not follow the expected program path. When all these options are turned on and all the above checks are satisfied, symbolic execution of an individual path has *perfect precision*: path constraint generation and solving is *sound and complete* (Sect. 2.1).

Moreover, we *turned off* all the *unsound state-space pruning* techniques and heuristics implemented in SAGE to limit path explosion, such as limiting the number of constraints generated for each program branch and constraint subsumption, which eliminates constraints logically implied by other constraints injected at the same program branch (most likely due to successive iterations of an input-dependent loop) using a cheap syntactic check [21]. How we dealt with path explosion in this work is discussed in Sect. 5.2 and 5.3.

As we describe in Sect. 5, we also used MicroX in conjunction with SAGE in order to prove memory safety of individual ANI functions in isolation. Memory safety of a function is *proven for any calling context* (soundly and completely) by MicroX and SAGE if all possible function input values are considered, symbolic execution of every function path is sound and complete, all function paths can be enumerated and tested in a finite (and small enough) amount of time, and all the checks defined above are satisfied for all executions. Instead of manually writing a unit test driver that explicitly identifies all input parameters (and their types) for each function, MicroX provided this functionality automatically [16].

During this work, many functions were not verified at first for various reasons: we discovered and fixed several x86 instructions unhandled by SAGE's symbolic execution engine, we also fixed several root causes of divergences (by providing custom summaries for nondeterministic-looking functions, like `malloc` and `memcpy`, whose execution paths depend on memory alignment), and we fixed a few imprecision bugs in SAGE's code. These SAGE limitations were much more easily identified when verifying small functions in isolation with MicroX rather than during whole-application fuzzing. After removing those limitations, we were able to verify that many individual ANI functions are memory safe (Sect. 5.1). The remaining functions could not be verified so easily mostly because of path explosion due to input-dependent loops (Sect. 5.2) or due to too many paths in functions lower in the callgraph (Sect. 5.3).

4 The ANI Windows Parser

The ANI Windows parser is written mostly in C, while the remaining code is written in x86 assembly. The implementation involves at least 350 functions defined in 5 Windows DLLs. The parsing of input bytes from an ANI file takes place in at least 110 functions defined in 2 DLLs, namely in `user32.dll`, which is responsible for 80% of the parsing code, and in `gdi32.dll`, which is responsible for the remaining 20%[4]. `user32.dll` creates and manages the Windows user

[4] These percentages were obtained by comparing the number of constraints on symbolic values that were generated by SAGE for each of the 2 DLLs.

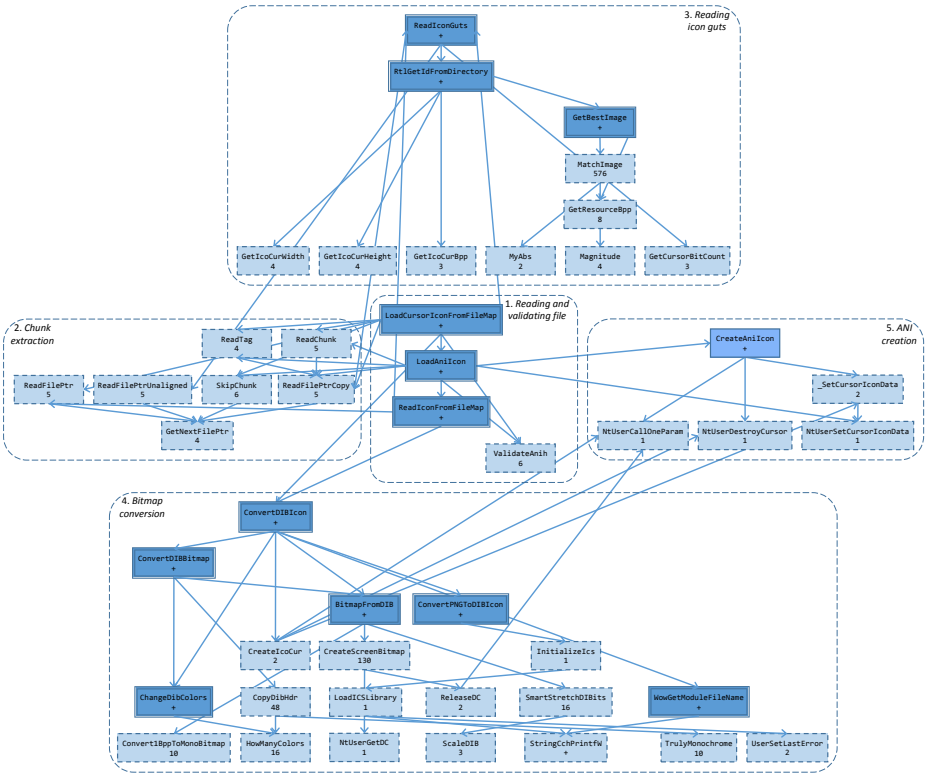

Fig. 1. The callgraph of the 47 `user32.dll` functions implementing the ANI parser core. Functions are grouped based on the architectural component of the parser to which they belong. The different shades and lines of the boxes denote the verification strategy we used to prove memory safety of each function. The boxes with the lighter shade and dotted lines indicate functions verified with the bottom-up strategy (Stage 1), the medium shade and single solid line functions verified by restricting the bounds of input-dependent loops (Stage 2), and the darker shade and double solid lines functions verified with the top-down strategy (Stage 3). Functions are annotated with the number of their execution paths. A + indicates that a function contains too many execution paths to be exhaustively enumerated within 12 hours without using additional techniques for controlling path explosion.

interface, such as windows, mouse events and menus. Many functions defined in `user32.dll` call into `gdi32.dll`, which is the graphics device interface associated with drawing and handling two-dimensional objects as well as managing fonts. There are 47 functions defined in `user32.dll` that implement functionality of the ANI parser. These functions alone compile to approximately 3,050 x86 instructions. More details on the ANI parser, including the file format it handles and its high-level callgraph, can be found in the appendix.

5 Verification Results

We proved memory safety of the ANI Windows image parser by targeting the 47 functions that are defined in `user32.dll` and are responsible for 80% of the

parsing code (Sect. 4). The remaining 20% refers to at least 63 `gdi32.dll` functions that are called (directly or indirectly) by the 47 `user32.dll` functions. In addition to those `user32.dll` and `gdi32.dll` functions, the parser also exercises code in at least 240 other functions (for a total of at least 350 functions). As shown by sound and complete symbolic execution, all these other functions do not (directly or indirectly) parse any input bytes from an ANI file and are by definition attacker memory safe. For the purpose of this work, the `gdi32.dll` and all these other functions can be viewed as *inlined* to the `user32.dll` functions, which are the top-level functions of the parser. Verifying those 47 `user32.dll` functions while inlining all remaining sub-functions is, thus, equivalent to proving attacker memory safety of the entire ANI parser. The callgraph of the 47 `user32.dll` functions is shown in Fig. 1. The functions are grouped depending on the architectural component of the parser to which they belong. Note that there is no recursion in this callgraph.

In this section, we describe how we proved memory safety of the ANI parser using compositional exhaustive testing. Our verification results were obtained with a 32bit Windows 7 version of the parser and are presented in three stages.

5.1 Stage 1: Bottom-Up Strategy

For verifying the ANI parser, we started with a bottom-up strategy with respect to the callgraph of Fig. 1. We wanted to know how many functions of a real code base can be proven memory safe for any calling context by simply using exhaustive path enumeration. Our setup for this verification strategy consisted in attempting to verify each `user32.dll` function (one at a time) using MicroX with SAGE starting from the bottom of the callgraph. If all execution paths of the function were explored in a reasonable amount of time, i.e., less than 12 hours, and no bugs or other incompleteness-check violations were ever detected (Sect. 3.3), we marked the function as memory safe. To our surprise, 34 of the 47 functions shown in Fig. 1 could already be proven memory safe this way, and are shown with the lighter shade and dotted lines in the figure.

An exception was the `StringCchPrintfW` function of the *Bitmap conversion* component. This function writes formatted data to a specified string, which is stored in a destination buffer. Exploring all execution paths of function `StringCchPrintfW` that may be passed a destination buffer of any length and a format string with any number of format specifiers does not complete in 12 hours, and is actually very complex.

Inlining. To deal with this function, we just *inlined* it to each of its callers. Inlining a function means replacing the call sites of the function with the function body. In our context, inlining a function means that the function being inlined is no longer treated as an isolated unit that we attempt to verify for any (all) calling contexts, but instead, it is being included in the unit defined by its caller function(s) and proven only for the specific calling context defined in these caller function(s). For instance, function `LoadICSLibrary`, which takes no input arguments, calls function `StringCchPrintfW`. By inlining `StringCchPrintfW` to `LoadICSLibrary`, we can exercise the single execution path in `LoadICSLibrary` and prove attacker memory safety of both functions.

Verification results. With the simple bottom-up strategy of this section, we were already able to prove attacker memory safety of 34 `user32.dll` functions out of 47, or 72% of the top-level functions of the ANI Windows parser. So far, we had to inline only one function, namely `StringCchPrintfW` to `LoadICSLibrary` of the *Bitmap conversion* component. The `gdi32.dll` functions (not shown in Fig. 1), which are called by the 47 `user32.dll` functions of Fig. 1, were also inlined (recursively) in those `user32.dll` functions. The boxes with the lighter shade and dotted lines of Fig. 1 represent the 34 functions that were verified with the bottom-up strategy. All these functions, except for those that were inlined, were verified in isolation for any calling context. This implies that all bounds for all loops (if any) in all those functions either do not depend on function inputs, or are small enough to be exhaustively explored within 12 hours. Recall that accesses to function input buffers are not yet proven memory safe (Sect. 3.2).

5.2 Stage 2: Input-Dependent Loops

For the remaining 13 `user32.dll` functions of the ANI parser, path explosion is too brutal and exhaustive path enumeration does not terminate in 12 hours. Therefore, during the second stage of the verification process, we decided to identify and restrict the bounds of *input-dependent loops* that might have been preventing us from verifying functions higher in the callgraph of the parser in Stage 1. We define an input-dependent loop as a loop whose number of iterations depends on bytes read from an ANI file, i.e., whole-program inputs. In contrast, when the number of iterations of a loop inside a function depends on function inputs that are not whole-program inputs, path explosion due to that loop can be eliminated by inlining that function to its caller(s).

Restricting input-dependent loop bounds. In order to control path explosion due to input-dependent loops, we manually *fixed the bounds*, i.e., the number of iterations, of those loops by assigning a concrete value to the program variable(s) containing the input bound(s). We extended MicroX for the user to easily fix the value of arbitrary x86 registers or memory addresses. Naturally, fixing an input value to a specific concrete value is like specifying an input precondition, and the verification of memory safety becomes restricted to calling contexts satisfying that precondition.

As an example, consider function `CreateAniIcon` of the *ANI creation* component of the parser. `CreateAniIcon` calls functions `NtUserCallOneParam` and `NtUserDestroyCursor`, which have one execution path each, as well as function `_SetCursorIconData`, which has two execution paths as shown in Fig. 1. Despite the very small number of paths in its callees, function `CreateAniIcon` contains too many paths to be explored in 12 hours, which is indicated by the + in Fig. 1. This path explosion is due to two input-dependent loops inside that function. By fixing the bounds of these loops to any value from 1 to 2^{32}, the number of execution paths in the loops of function `CreateAniIcon` is always 4. Thus, we can prove memory safety of `CreateAniIcon` for any such fixed number of iterations of these loops. More details can be found in the appendix.

Verification results. During this stage of the verification process, we proved memory safety of only one additional `user32.dll` function of the ANI parser,

Type of loop bound	Component	Maximum loop bound
Frames (4 bytes)	5	2^{32}
Steps (4 bytes)	5	2^{32}
Images/frame (2 bytes/frame)	3	1
File size	1	110

Tab. 1. All the input-dependent loop bounds fixed during the verification of the ANI parser. For each loop bound, the table shows the corresponding number of bytes in an ANI input file, the component of the parser containing loops with this bound (numbered as in Fig. 1), and the maximum value of the bound that we could verify in 12 hours.

namely `CreateAniIcon`. The box in Fig. 1 with the medium shade and single solid line represents function `CreateAniIcon` that was verified in Stage 2.

Tab. 1 presents a complete list of the input-dependent loop bounds that we fixed during the entire verification of the ANI parser. As described above, to verify memory safety of function `CreateAniIcon` of the *ANI creation* component (component 5 of Fig. 1), we had to fix two input-dependent loops using two whole-program input parameters (namely, frames and steps). In the remainder of this work (Sect. 5.3), we also had to fix two other whole-program input parameters to control a few other input-dependent loops. First, in the *Reading icon guts* component (component 3 of Fig. 1), there are three other input-dependent loops, located in functions `ReadIconGuts` and `GetBestImage`. The number of iterations of all those loops depends on the number of images contained in each icon, which corresponds to 2 bytes per frame of an ANI file. (A single icon may consist of multiple images of different sizes and color depths.) To limit path explosion due to those three loops, we had to fix the number of images per icon of the animated cursor to a maximum of 1. Second, in the *Reading and validating file* component (component 1 of Fig. 1), there are two input-dependent loops, located in functions `LoadCursorIconFromFileMap` and `LoadAniIcon`, whose number of iterations depends on the size of the input file, which we had to restrict to a maximum of 110 bytes.

It is perhaps surprising that the number of input-dependent loop bounds in the entire parser is limited to a handful of input parameters read from an ANI file, for a total of around 10 bytes (plus the input file size) as shown in Tab. 1.

5.3 Stage 3: Top-Down Strategy

For the remaining 12 `user32.dll` functions still to be verified in the higher-level part of the callgraph of Fig. 1, path explosion was still too severe even after using inlining and fixing input-dependent loops. Therefore, we adopted a different, top-down strategy using sub-function summaries in order to prove memory safety compositionally as described in Sect. 2.2 and 3.

Summarization. As we explained earlier, summarizing sub-functions can alleviate path explosion in those sub-functions at the expense of computing reusable logic summaries that capture function pre- and postconditions expressed

in terms of function inputs and outputs, respectively. For this trade-off to be attractive, it is therefore best to summarize sub-functions (1) that contain many execution paths and (2) whose input/output interfaces with respect to higher-level functions are not too complex so that the logic encoding of their summaries remains tractable. Moreover, to prove memory safety of a sub-function with respect to its input buffers, all bounds-checking constraints inside that sub-function must be included in the precondition of its summary (Sect. 3.2).

Verification results. To verify the remaining 12 top-level `user32.dll` functions, we manually devised the following summarization strategy based on the previous data about the numbers of paths in verified sub-functions (i.e., the numbers of paths in the boxes of Fig. 1) and by examining the input/output interfaces of the remaining functions. Specifically, we verified one by one the top-level function of each remaining component of the parser, namely function `ReadIconGuts` of the *Reading icon guts* component, `ConvertDIBIcon` of the *Bitmap conversion* component, and `LoadCursorIconFromFileMap` of the *Reading and validating file* component as follows (since the *Chunk extraction* and *ANI creation* components had already been verified during the previous stages).

Verification of `ReadIconGuts`. (*Reading icon guts* component) We fixed the bounds of the input-dependent loops of this component to a single loop iteration (Tab. 1) as discussed in Sect. 5.2, and summarized function `MatchImage`. This function only returns an integer (a "score") that does not influence the control-flow execution of its caller `GetBestImage` for one loop iteration, so its visible postcondition $post_f$ is very simple. Moreover, `MatchImage` takes only one buffer as input, therefore the precondition of its summary includes only bounds-checking constraints for that buffer. In its caller `GetBestImage`, the size of this buffer is always constant and equal to the size of a structure, so `MatchImage` is attacker memory safe. Overall, when restricting the bounds of the input-dependent loops in the *Reading icon guts* component, summarizing `MatchImage`, and inlining all the other functions below it in the callgraph, `ReadIconGuts` contained 468 execution paths that are explored by our tools in 21m 53s.

Verification of `ConvertDIBIcon`. (*Bitmap conversion* component) In a similar way, we verified this function after summarizing sub-function `CopyDibHdr`, whose summarization is also tractable in practice (details not shown here). After summarization, `ConvertDIBIcon` contains 28 execution paths exercised in 1m 58s. Note that, in the *Bitmap conversion* component, there are no input-dependent loops; although sub-function `ConvertPNGToDIBIcon` has loops whose numbers of iterations depend on this function's inputs and therefore could not be verified in isolation, inlining it to its caller `ConvertDIBIcon` eliminated this source of path explosion and it was then proven to be attacker memory safe.

Verification of `LoadCursorIconFromFileMap`. (*Reading and validating file* component) This is the very top-level function of the parser and the final piece of the verification puzzle. Since this final step targets the verification of the entire parser, it clearly requires the use of summarization to alleviate path explosion.

Fortunately, and perhaps surprisingly, after closely examining the implementation of the ANI parser's components, we realized that it is common for their

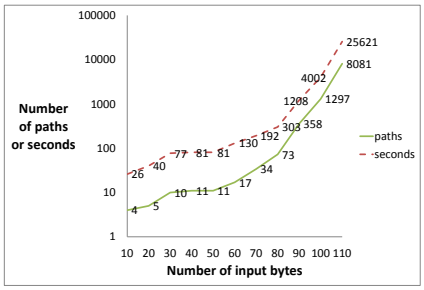

Fig. 2. The number of execution paths in the top-level function `LoadCursorIconFromFileMap` of the ANI parser and the time (in seconds) it takes to exercise these paths versus the number of input bytes when summarizing components *Reading icon guts*, *Bitmap conversion*, and *ANI creation*.

output to be a single "success" or "failure" value. In case "failure" is returned, the higher-level component typically terminates. In case "success" is returned, the parsing proceeds but without reading any other sub-component outputs and with reading other higher-level inputs (such as other bytes that follow in the input file), i.e., completely independently of the specific path taken in the sub-component being summarized. Therefore, the visible postcondition of function summaries with such interfaces is very simple: a success/failure value. This is the case for the top-level functions of the lower-level components *Reading icon guts*, *Bitmap conversion*, and *ANI creation*. This was not the case for the *Chunk extraction* component, which mainly consists of auxiliary functions but does not significantly contribute to path explosion and was not summarized.

More specifically, for the verification of `LoadCursorIconFromFileMap`, we used three summaries for the following top-level functions of sub-components:

- `ReadIconGuts`, which returns a pointer to a structure that is checked for nullness in its callers. Then, caller `LoadCursorIconFromFileMap` returns null when this pointer is null. In caller `ReadIconFromFileMap`, in case the pointer is non-null, it is passed as argument to `ConvertDIBIcon`, which has already been verified for any calling context as described above.
- `ConvertDIBIcon`: case similar to `ReadIconGuts`.
- `CreateAniIcon`, which also returns a pointer to a structure. If this pointer is null, the parser fails and caller `LoadAniIcon` emits an error message:

```
if (frames != 0) ani = CreateAniIcon(...);
if (ani == NULL) EMIT_ERROR("Invalid icon");
```

  Otherwise, the pointer is returned by `LoadAniIcon` and subsequently by the top-level function of the parser.

Function `LoadCursorIconFromFileMap` also has an input-dependent loop whose number of iterations depends on the size of the input file being read and containing the ANI file to be parsed. By summarizing the top-level function of the above three lower-level components and fixing the file size, we were able to prove memory safety of the parser up to a file size of 110 bytes in less than 12 hours. Fig. 2 shows the number of execution paths in the parser as well as the time it takes to explore these paths when summarizing components *Reading icon guts*, *Bitmap conversion*, and *ANI creation* and controlling the file size.

6 Memory-Safety Bugs

In reality, the verification of the ANI Windows parser was slightly more complicated than presented in the previous section because the ANI parser is actually *not memory safe*! Specifically, we found three types of memory-safety violations during the course of this work:

- real bugs (fixed in the latest version of Windows),
- harmless bugs (off-by-one non-exploitable buffer overflows),
- code parts not memory safe by design.

We briefly discuss each of these memory-safety violations. Details are omitted on purpose.

Real bugs. We found several buffer overflows all related to the same root cause. Function `ReadIconGuts` of the *Reading icon guts* component allocates memory for storing a single icon extracted from the input file and returns a pointer to this memory. The allocated memory is then cast to a structure, whose fields are read for accessing sub-parts of the icon, such as its header. However, the size of an icon, and therefore the size of the allocated memory, depends on the (untrusted) declared size of the images that make up the icon. These sizes are declared in the ANI file and might not correspond to the actual image sizes. Consequently, if the declared size of the images is too small, then the size of the allocated memory is too small, and there are buffer overflows when accessing the fields of the structure located beyond the allocated memory for the icon. These buffer overflows have been fixed in the latest version of Windows, but are believed to be hard to exploit and hence not security critical.

Harmless bugs. We also found several harmless buffer overflows related to the bugs described above. For instance, function `ConvertPNGToDIBIcon` of the *Bitmap conversion* component converts an icon in PNG format to DIB (Device Independent Bitmap), and also takes as argument a pointer to the above structure for the icon. To determine whether an icon is in PNG format, `ConvertPNGToDIBIcon` checks whether the icon contains the 8-byte PNG signature. However, the allocated memory for the icon may be smaller than 8 bytes, in which case there can be a buffer overflow. Still, on Windows, every memory allocation (call to `malloc`) always results in the allocation of a reserved memory block of at least 8 bytes. So technically, accessing any buffer `buf` of size less than 8 up to `buf+7` bytes is not a buffer overflow according to the Windows runtime environment—such buffer overflows are harmless to both reliability and security.

Code parts not memory safe by design. Finally, we found memory-safety violations that were expected and caught as runtime exceptions using `try/except` statements. For instance, `CopyDibHdr` of the *Bitmap conversion* component copies and converts an icon header to a common header format. The size of the memory that is allocated in `CopyDibHdr` for copying the icon header depends on color information defined in the header itself. This color information is read from the input file, and is therefore untrusted. Specifically, it can make the parser allocate a huge amount of memory, which is often referred to as a *memory spike*. Later, the actual header content is copied into this memory. To check whether the declared size matches the actual size, `CopyDibHdr` uses a `try`

statement to probe the icon header in chunks of 4K bytes, i.e., the minimum page size, to ensure that the memory is readable and properly initialized. While probing the icon header inside the `try` statement, the parser may access unallocated memory beyond the bounds of the header, which is a memory-safety violation. However, this violation is expected to be caught in an `except` statement, which aborts parsing in higher-level functions.

The verification results of Sect. 5 were obtained after fixing or ignoring the memory-safety bugs discussed in this section. Those results are therefore sound only with respect to these additional assumptions.

7 Other Related Work

Traditional *interactive* program verification, based on static program analysis, verification-condition generation, and theorem proving, provides a broader framework for proving more complex properties of a larger class of programs but at the expense of more work from the user. For instance, the VCC [12] project verified the *functional correctness*, including memory safety and race freedom, of the Microsoft Hyper-V hypervisor [27], a piece of concurrent software (100K lines of C, 5K lines of assembly), and required more than 13.5K lines of source-code annotations for specifying contracts, loop invariants, and ghost state in about 350 functions by a team of more than 10 people and over a period of several years. As another impressive example, the seL4 project [25] designed and verified the C code of a microkernel using the interactive theorem prover Isabelle/HOL [31] and requiring about 200K lines of Isabelle scripts and 20 years of research in developing and automating the proofs. Also recently, Typed Assembly Language [29] (TAL) and the Boogie program verifier [4] were used to prove type and memory safety of (part of) the Verve operating system [37] (a total of 20 functions implemented in approximately 1.5K lines of x86 assembly), manually annotated with pre-/postconditions, loop invariants, and external function stubs for a total of 1,185 lines of annotations in about nine months of work.

In contrast, our verification project required only three months of work, no program annotations, no static program analysis, and no external function stubs, although our scope was more focused (attacker memory safety only), our application domain was different (sequential image parser versus concurrent/reactive operating-system code), and we did require several key manual verification steps, including fixing a few input-dependent loop bounds, as discussed in Sect. 5. Note that our purely dynamic techniques and x86-based tools can handle ANI x86 code patterns such as stack-modifying compiler-injected code for structured exception handling (SEH prologue and epilogue code for `try/except` statements) and stack-guard protection, which most static-analysis tools cannot handle.

Static-analysis-based software model checkers, like SLAM [3], BLAST [23], and Yogi [32], can *automatically* prove control-oriented API properties of specific classes of programs (specifically, device drivers). These tools rely on (predicate) abstraction in order to scale, and are not engineered to reason precisely about pointers, memory alignment, and aliasing. They were not designed and cannot be used as-is for proving (attacker) memory safety of an application as large and complex as the ANI Windows parser.

SAT/SMT-based bounded model checkers, as CBMC [11], are another class of static-analysis tools for automatic program verification. For loop-free programs and when symbolic execution has perfect precision, the program's logic representation generated by such model checkers is similar to verification-condition generation and captures both data and control dependencies on all program variables, which is similar to *eagerly* summarizing (as in Sect. 2.2) *every* program block and function. Even excluding all loops, such a monolithic whole-program logic encoding would not scale to accurately represent the entire ANI parser.

As shown in Sect. 5, systematic dynamic test generation also does not scale to the entire ANI parser without the selective use of function summarization and fixing a few input-dependent loop bounds. These crucial steps were performed manually in our work. Algorithms and heuristics for *automatic* program summarization have been proposed before [15, 1, 26] as well as other closely related techniques [5, 28] and heuristics [21], which can be viewed as approximations of sub-program summarization. However, none of this prior work on automatic summarization has ever been applied to *verify* an application as large and complex as the parser considered here.

We emphasize that we are not aware of *any* automatic tool that, today, could prove (attacker) memory safety of an application like the ANI parser. We do not know which parts of the ANI code are in the subset of C for which tools like CCured [30] or Prefix [7] are sound, or how many memory-safety checks could be removed in those parts with such a sound static analysis. However, we do know that Prefix was run on this code for years, yet bugs remained, which is precisely why fuzzing is performed later [6].

Proving *attacker memory safety*, even more so *compositionally*, is novel: we prove that an attacker cannot trigger buffer overflows, but ignore other buffer overflows (for instance, due to the failure of trusted system calls). This requires a whole-program taint analysis to focus on what the attacker can control, performed using symbolic execution and the top-down strategy of Sect. 5.3. In contrast, other approaches like verification-condition generation, bounded model checking or traditional static analysis lack this global taint view and treat all program statements alike, without prioritizing the analysis towards parts closest to the attack surface, which hampers scalability and relevance to security.

8 Concluding Remarks

We showed how to prove attacker memory safety of an entire operating-system image parser using compositional exhaustive testing, i.e., no static analysis whatsoever. These results required a high-level of automation in our tools and verification process although key steps were performed manually, like fixing input-dependent loop bounds, guiding the summarization strategy, and fixing and avoiding memory-safety violations. Also, the scope of our work was only to prove attacker memory safety, not general memory safety or functional correctness, and the ANI parser is a purely sequential program. Finally, the verification guarantees provided by our work are valid only with respect to some important assumptions we had to make, mostly regarding input-dependent loop bounds. Overall, after this work, we are now confident that the presence of any remaining

security-critical (i.e., attacker-controllable) buffer overflows in the ANI Windows parser is unlikely, but those conclusions are subject to the assumptions we made.

Here are some interesting findings that we did not expect:

- many ANI functions are loop free and were easy to verify (Sect. 5.1);
- all the input-dependent loops in the entire ANI parser are controlled by the values of about 10 bytes only in any ANI file plus the file size (Sect. 5.2);
- the remaining path explosion can be controlled by using only 5 function summaries with very simple interfaces (Sect. 5.3).

Our work suggests future directions for automating further several of the steps that were done manually (e.g., dealing with few but critical input-dependent loops and program decomposition at cost-effective interfaces). Perhaps future tools will perform those steps intelligently and automatically.

References

1. S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *TACAS*, volume 4963 of *LNCS*, pages 367–381. Springer, 2008.
2. S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. M. Paradkar, and M. D. Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *TSE*, 36:474–494, 2010.
3. T. Ball and S. K. Rajamani. The SLAM toolkit. In *CAV*, volume 2102 of *LNCS*, pages 260–264. Springer, 2001.
4. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005.
5. P. Boonstoppel, C. Cadar, and D. R. Engler. RWset: Attacking path explosion in constraint-based test generation. In *TACAS*, volume 4963 of *LNCS*, pages 351–366. Springer, 2008.
6. E. Bounimova, P. Godefroid, and D. A. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *ICSE*, pages 122–131. ACM, 2013.
7. W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *SPE*, 30:775–802, 2000.
8. C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224. USENIX, 2008.
9. C. Cadar and D. R. Engler. Execution generated test cases: How to make systems code crash itself. In *SPIN*, volume 3639 of *LNCS*, pages 2–23. Springer, 2005.
10. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *CCS*, pages 322–335. ACM, 2006.
11. E. M. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *DAC*, pages 368–371. ACM, 2003.
12. E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.
13. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
14. B. Elkarablieh, P. Godefroid, and M. Y. Levin. Precise pointer reasoning for dynamic test generation. In *ISSTA*, pages 129–140. ACM, 2009.
15. P. Godefroid. Compositional dynamic test generation. In *POPL*, pages 47–54. ACM, 2007.

16. P. Godefroid. Micro execution. In *ICSE*, pages 539–549. ACM, 2014.
17. P. Godefroid and J. Kinder. Proving memory safety of floating-point computations by combining static and dynamic program analysis. In *ISSTA*, pages 1–12. ACM, 2010.
18. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, pages 213–223. ACM, 2005.
19. P. Godefroid, S. K. Lahiri, and C. Rubio-González. Statically validating must summaries for incremental compositional dynamic test generation. In *SAS*, volume 6887 of *LNCS*, pages 112–128. Springer, 2011.
20. P. Godefroid, M. Y. Levin, and D. A. Molnar. Active property checking. In *EMSOFT*, pages 207–216. ACM, 2008.
21. P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS*, pages 151—166. The Internet Society, 2008.
22. P. Godefroid and D. Luchaup. Automatic partial loop summarization in dynamic test generation. In *ISSTA*, pages 23–33. ACM, 2011.
23. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70. ACM, 2002.
24. M. Howard. Lessons learned from the animated cursor security bug, 2007. `http://blogs.msdn.com/b/sdl/archive/2007/04/26/lessons-learned-from-the-animated-cursor-security-bug.aspx`.
25. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *SOSP*, pages 207–220. ACM, 2009.
26. V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *PLDI*, pages 193–204. ACM, 2012.
27. D. Leinenbach and T. Santen. Verifying the Microsoft Hyper-V hypervisor with VCC. In *FM*, volume 5850 of *LNCS*, pages 806–809. Springer, 2009.
28. R. Majumdar and R.-G. Xu. Reducing test inputs using information partitions. In *CAV*, volume 5643 of *LNCS*, pages 555–569. Springer, 2009.
29. J. G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. In *POPL*, pages 85–97. ACM, 1998.
30. G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *POPL*, pages 128–139. ACM, 2002.
31. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL—A proof assistant for higher-order logic*, volume 2283 of *LNCS*. Springer, 2002.
32. A. V. Nori and S. K. Rajamani. An empirical study of optimizations in YOGI. In *ICSE*, pages 355–364. ACM, 2010.
33. K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *CAV*, volume 4144 of *LNCS*, pages 419–423. Springer, 2006.
34. D. X. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *ICISS*, volume 5352 of *LNCS*, pages 1–25. Springer, 2008.
35. A. Sotirov. Windows animated cursor stack overflow vulnerability, 2007. `http://www.offensive-security.com/os101/ani.htm`.
36. N. Tillmann and J. de Halleux. Pex—White box test generation for .NET. In *TAP*, volume 4966 of *LNCS*, pages 134–153. Springer, 2008.
37. J. Yang and C. Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *PLDI*, pages 99–110. ACM, 2010.

A The ANI Windows Parser

The ANI Windows parser handles a structured graphics file format for reading and storing animated cursors like the spinning ring or hourglass on Windows. Such animated icons are ubiquitous in practice, and their domain of use ranges from web pages and blogs, instant messaging and e-mails, to presentations and video clips. In addition, there are many applications for creating, editing, and converting these icons to and from different file formats, such as GIF or CUR.

We chose to prove memory safety of this ANI parser as it is one of the smallest image parsers embedded in Windows. The implementation of the parser is also within the scope of our tools since it is neither concurrent nor subject to real-time constraints. Despite this, there are still significant challenges in proving memory safety of the ANI parser including reasoning about memory dereferences and exception handling code. Our choice was also motivated by the fact that in 2007 a critical out-of-band security patch was released for code in this parser (MS07-017) costing Microsoft and its users millions of dollars. This vulnerability was similar to an earlier one reported in 2005 (MS05-002) meaning that many details of the ANI parser have already been made public over the years [35, 24]. This parser is included in all distributions of Windows, i.e., it is used on more than a billion PCs, and has been fuzzed for years (with SAGE among other tools). Given the ubiquity of animated icons, our goal was to determine whether the ANI parser is now free of security-critical buffer overflows.

The general format of an ANI file is shown in Fig. 3. It is based on the generic Resource Interchange File Format (RIFF) for storing various types of data in tagged chunks, such as video (AVI) or digital audio (WAV). RIFF has a hierarchical structure in which each chunk might contain data or a list of other chunks. Animated icons contain the following information:

- a `RIFF` chunk, whose header has the identifier `ACON`, specifies the type of the file,
- an optional `LIST` chunk, whose header has the identifier `INFO`, contains information about the file, such as the name of the artist,
- an `anih` header chunk contains information about the animation including the number of frames, i.e., Windows icons, and the number of steps, i.e., the total number of times the frames are displayed,
- an optional `seq` chunk defines the order in which the frames are displayed,
- an optional `rate` chunk determines the display rate for each frame in the sequence, and
- a `LIST` chunk, whose header has the identifier `fram`, contains a list of icons.

This file format already provides an indication of the size and complexity of the ANI parser.

The high-level callgraph of the parser code is shown in Fig. 4. The main component of the architecture, *Reading and validating file*, reads and validates each chunk of an ANI file. If an extracted chunk is a `LIST fram`, the *Reading icon guts* component is invoked to read and validate the first icon in the list. In case the icon is valid, it is converted to a physical bitmap object by the *Bitmap conversion* component. Once this process has been repeated for all the icons

```
RIFF ACON
  [ LIST INFO
      IART <artist>
      ICOP <copyright>
      INAM <name>
  ]
  anih <anihdr>
  [ seq <seqinfo> ]
  [ rate <rateinfo> ]
  LIST fram icon <iconfile> ...
```

Fig. 3. ANI file format (partial description).

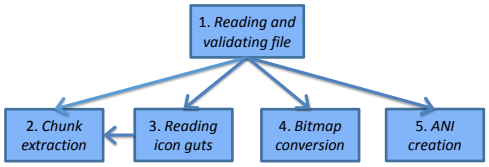

Fig. 4. High-level callgraph of the ANI parser.

```
for (i = 0; i < frames; i++) frameArrT[i] = frameArr[i];
for (i = 0; i < steps; i++) {
  if (rateArr == NULL) rateArrT[i] = rate;
  else rateArrT[i] = rateArr[i];
  if (stepArr == NULL) stepArrT[i] = i;
  else stepArrT[i] = stepArr[i];
}
```

Fig. 5. The input-dependent loops in function `CreateAniIcon` (code fragment). Variables `frames`, `steps`, `rateArr`, and `stepArr` are inputs to `CreateAniIcon`. All arrays are allocated such that their length is greater than the corresponding loop bound, and therefore, there are no buffer overflows in this code.

in the list, the animated icon is created from their combination (*ANI creation* component).

B Input-Dependent Loops in Function `CreateAniIcon`

The path explosion in function `CreateAniIcon` is due its input-dependent loops shown in Fig. 5. The loop bounds `frames`, which refers to the number of frames in an animated cursor, and `steps`, which refers to the number of steps, are both inputs to `CreateAniIcon`, and so are the values of variables `rateArr` and `stepArr`. Since `frames` and `steps` are of type `int` (4 bytes), each loop may iterate up to 2^{32} times, which leads to the exploration of 2^{32} possible execution paths in `CreateAniIcon`, and is intractable in practice. Consequently, to control path explosion and verify this function, we fixed the values of `frames` and `steps`. For any fixed value of `frames`, the first loop of Fig. 5 has only 1 execution path, while for any fixed value of `steps`, the second loop has always 4 execution paths due to the tests on the other inputs `rateArr` and `stepArr`. Thus, by fixing these loop bounds to any value from 1 to 2^{32}, the number of execution paths in the loops of Fig. 5 is always 4. Tab. 2 summarizes how the number of paths in

Input values		Number of paths
`frames`	`steps`	
0	0	1
1	0	1
any fixed	0	1
0	1	4
0	any fixed	4
any fixed	any fixed	4

Tab. 2. The number of paths in the loops of `CreateAniIcon` (shown in Fig. 5) changes when fixing the input-dependent loop bounds `frames` and `steps` to different values.

the loops of `CreateAniIcon` changes when fixing `frames` and `steps` to different values. As Tab. 2 shows, we can prove memory safety of function `CreateAniIcon` for any fixed number of frames and steps in an animated cursor.