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

Maria Christakis
Department of Computer Science
ETH Zurich, Switzerland
maria.christakis@inf.ethz.ch

Patrice Godefroid
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 [17, 8] 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 [9], jCUTE [31], SAGE [20], Pex [34], KLEE [7], BitBlaze [32], 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 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 in order to prove memory safety of the ANI Windows image parser. This parser is responsible for reading files in a structured graphics file format, and processing their contents in order 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 purely 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 (Sect. 4). One of the motivations for this work was to determine whether the ANI Windows parser is now free of security-critical buffer overflows.

We show in this paper 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 along all the explored code paths 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:
• We report on the first application of systematic dynamic test generation for verifying a real, complex, security-critical, entire program.
• To the best of our knowledge, this is the first time that an operating-system (Windows or other) image parser has ever been proven free of security-critical buffer overflows.
• 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 [17, 8] 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.

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 φw is a logic formula that characterizes the input values for which the program executes along w. Each symbolic variable appearing in φw is, thus, a program input. Each constraint is expressed in some theory T decided by a constraint solver, i.e., an automated theorem prover that can be alleviated by performing symbolic execution compositionally [14, 1].

All program paths can be enumerated by a search algorithm that explores all possible branches at conditional statements. The paths w for which φw is satisfiable are feasible, and are the only ones that can be executed by the actual program provided the solutions to φw characterize exactly the inputs that drive the program through w. Assuming that the constraint solver used to check the satisfiability of all formulas φw 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 [14, 1].

In compositional symbolic execution, a summary φf for a function (or any program sub-computation) f is defined as a logic formula over constraints expressed in theory T. Summary φf 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. More precisely, φf is a conjunction of constraints on inputs of f, and 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 an output of f is any value written by f. Therefore, φw is the constraint path along path w but expressed in terms of the function inputs, while φw 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 (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. At the end of the execution of w, 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,

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

a summary φf for this function can be

φf = (x > 0 ∧ ret = 1) ∨ (x ≤ 0 ∧ 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 φfp symbolically representing all the paths discovered so far during the search. By construction [14], symbolic variables corresponding to function inputs and outputs are all bound in φfp, 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,

```c
#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(); // (*) return;
}
```

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

\[
\bigwedge_{0 \leq i < N} ((s[i] > 0 \land ret_i = 1) \lor (s[i] \leq 0 \land ret_i = 0))
\]

where reti denotes the return value of the ith call to function is_positive. Even though program P has 2N 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 \texttt{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., [14, 1, 5, 25]), 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 [20] and MicroX [15]. 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 [18] as well as specialized forms of summaries for dealing with floating-point computations [16] and input-dependent loops [21]. 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 in a different manner as explained in Sect. 5.2.

MicroX is a newer tool [15] 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. (Note that under the default memory policy values read from uninitialized global variables are not considered inputs.) 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 program 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 is stored in a program variable \texttt{v} (i.e., \texttt{a} = \texttt{M(v)}) is accessed during execution, the concrete value \texttt{a} of the address is first checked "passively" to make sure it points to a valid memory region \texttt{mr}_a as done in standard tools like Purify, Valgrind and Ap-
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:

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

If `foo` calls `bar` with `x=5`, the precondition of the above path summary for `bar` is satisfied. The bounds-checking constraint can be simplified with `mvsize = 5` in this calling context and negated to obtain the new path constraint,

\[
(0 \leq x) \land (x < 10) \land \neg (0 \leq x < 5)
\]

which after simplification is

\[
(0 \leq x) \land (x < 10) \land ((x < 0) \lor (x \geq 5))
\]

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 pre-condition 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.

### 3.3 Verification with SAGE and MicroX

We have implemented in SAGE the compositional procedure for proving memory safety described in Sect. 3.2. 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 [13], 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 [12], 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 then 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 [20]. 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 taken into account, 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 [15].

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 these 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 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 [33, 23]. 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. 1. It is based on the generic Resource Interchange Format (RIFF) for storing various types of data in tagged chunks,
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 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 ANI 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 call-graph of the 47 user32.dll functions is shown in Fig. 3. The functions are grouped depending on the component of Fig. 2 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 ANI 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. 3. 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 and 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. 3 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. It takes as input arguments the destination buffer that receives the formatted string, the size of the destination buffer, the format string, and the arguments that are inserted in the format string. 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 callee function(s). For instance, function LoadICSLibrary, which takes no input arguments, calls
Figure 3: The callgraph of the 47 user32.dll functions implementing the ANI parser core. Functions are grouped based on the architectural component of Fig. 2 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.

Verification results. With the simple bottom-up verification 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. 3), which are called by the 47 user32.dll functions of Fig. 3, were also inlined (recursively) in those user32.dll functions. The boxes with the lighter shade and dotted lines of Fig. 3 represent the 34 functions that were verified with the bottom-up strategy. It is important to note that all these functions, except for those that were inlined, were verified in isolation for any calling context. Recall that “verified in isolation” means that accesses to function input buffers are not yet proven memory safe at this stage of the verification process since input buffer sizes are still unknown (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, and _SetCursorIconData, which has two execution paths as shown in Fig. 3. 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. 3. This path explosion is due to the input-dependent loops inside that function shown in Fig. 4. 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. 4 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 steps path, while for any fixed value of frames, the first loop of Fig. 4 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 CreateAniIcon changes when fixing frames and steps to different values. As Tab. 1 shows, we can prove memory safety of function CreateAniIcon for any fixed number of frames and steps in an animated cursor.

Verification results. During this stage of the verification process, we proved memory safety of only one additional user32.dll function of the ANI parser, namely of CreateAniIcon. The box in Fig. 3 with the medium shade and single solid line represents function CreateAniIcon that was verified in Stage 2.

Tab. 2 presents a complete list of the input-dependent loop bounds that we fixed during the entire verification of the ANI parser. As we just explained, we had to fix two input-dependent loops using two whole-program input parameters (namely, frames and steps) in order to be able to verify memory safety of function CreateAniIcon of the ANI creation component (component 5 of Fig. 3).

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. 3), 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. 3), 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.

In summary, it is perhaps surprising that the number of input-dependent loop bounds in the entire ANI 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. 2.

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. 3, 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 re-usable 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...
Verifying 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. 3) 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) We fixed the bounds of the input-dependent loops of this component to a single loop iteration (Tab. 2) 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) 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) This is the very top-level function of the entire parser considered in this work and the final piece of the verification puzzle. Since this final step targets the verification of the entire parser implementation, 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 (Fig. 2), we realized that it is common for their 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 execution and parsing proceed 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 the top-level function LoadCursorIconFromFileMap, we used summaries for the following top-level functions of sub-components:

- Function ReadIconGuts returns a pointer to a structure that is checked for nullness in its callers. Then, function LoadCursorIconFromFileMap returns null when this pointer is null. In function 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 also returns a pointer to a structure. If this pointer is null, the parser fails and LoadANIIcon emits an error message as shown below:
  ```c
  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 includes 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. 5 shows the number of execution paths in the ANI parser as well as the time it takes to explore all these paths when summarizing components Reading icon guts, Bitmap conversion, and ANI creation and controlling the size of the input file.

With this top-down verification strategy and the careful use of function summarization, we were able to prove memory safety of the remaining 12 user32.dll functions of the ANI parser. Note that by inlining functions that were previously verified in isolation, we also proved that accesses to input buffers of these functions are memory safe. The boxes in Fig. 3 with the darker shade and double solid lines represent

<table>
<thead>
<tr>
<th>Type of loop bound</th>
<th>Component</th>
<th>Maximum loop bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frames (4 bytes)</td>
<td>5</td>
<td>$2^{32}$</td>
</tr>
<tr>
<td>Steps (4 bytes)</td>
<td>5</td>
<td>$2^{32}$</td>
</tr>
<tr>
<td>Images/frame (2 bytes/frame)</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>File size</td>
<td>1</td>
<td>110</td>
</tr>
</tbody>
</table>

Table 2: 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 that contains loops with this bound (numbered as in Fig. 3), and the maximum value of the bound that we could verify in 12 hours.
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 discuss each of these memory-safety violations in this section. Details are omitted on purpose. The verification results presented in Sect. 5 were actually obtained after fixing or ignoring these bugs as explained below.

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 probes the icon header in chunks of 4K bytes, i.e., the minimum page size, to ensure that the memory is readable and properly initialized as shown below:

```
try {
    DWORD offset;
    for (offset = 0; offset < alloc; offset += 0x1000)
        *(volatile BYTE *) ((LPBYTE)hdr + offset);
} except (W32ExceptionHandler(FALSE, RIP_WARNING)) {
    return NULL;
}
```

In the code above, variable alloc is the untrusted declared size, while variable hdr is a pointer to a buffer whose size is the real header size. 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 the except statement, which then simply returns NULL, which in turn aborts parsing in higher-level functions.

In order to determine whether these buffer overflows were real bugs, we identified the position of the (untrusted) input bytes that were to blame for the bugs in a well-formed input file. We subsequently changed the values of these bytes in the well-formed input file to the "buggy" values we had previously found, and then ran the entire ANI parser on the modified file. We could then witness that these buffer overflows were still triggered, hence proving that there was no input validation on the modified input bytes anywhere else in the ANI parser, and that these buffer overflows were reproducible and not false alarms. Note that we found no false positives during this work: all the buffer overflows we detected were indeed due to accesses to unallocated memory.

In a similar way, we were able to demonstrate that code surrounding the try/except code pattern shown above could be tricked in two different places into allocating 1MB and 1.5GB of memory, respectively, in function ReadIconGuts; in both cases though, the memory is freed before the function returns, so memory spikes are not observable.

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. For example, in the try/except code pattern above, variable alloc is input dependent and the for loop is an input-dependent loop as defined in Sect. 5.2 causing severe path explosion. To avoid path explosion due to such memory-unsafe code patterns, we restricted the values that variables like alloc can take.

---

**Figure 5:** 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.
Traditional interactive program verification, using static program analysis, verification-condition generation, and theorem proving, provides a much 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 [11] project verified the functional correctness, including memory safety and race freedom, of the Microsoft Hyper-V hypervisor [26], a relatively thin layer of concurrent software (100K lines of C, 5K lines of assembly) that runs between x64 hardware and guest operating systems and provides isolated execution environments, called partitions; this verification effort 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 [24] designed and verified the C code of a microkernel using the interactive theorem prover Isabelle/HOL [29] and requiring about 200K lines of Isabelle scripts and 20 years of research in developing and automating the proofs. Also recently, Typed Assembly Language [28] (TAL) and the Boogie program verifier [4] were used to prove type and memory safety of the Verve operating system [35], which consists of a low-level “Nucleus” written in x86 assembly and a higher-level kernel written in safe C/C++. The exported functions of the Nucleus code (a total of 20 functions implemented in approximately 1.5K lines of x86 assembly) were verified and 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 the bounds of a few input-dependent loops, as discussed in detail in Sect. 5. Note that our purely dynamic techniques and x86-based tools were able to deal with 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-overflow protection, which most static-analysis approaches do not handle. Static-analysis-based software model checkers, like for instance SLAM [3], BLAST [22], and Yogi [30], 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 [10], 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 also 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 Windows 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 [14, 1, 25] as well as other closely related techniques [5, 27] and heuristics [20], 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 ANI Windows parser.

7. OTHER RELATED WORK

For the first time, we were able to prove attacker memory safety of an entire operating-system image parser in only three months of work 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 at the beginning of this project:

- 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).

In hindsight, there are also several things we would now do differently. Mostly, the verification results obtained for the lower-level functions with the bottom-up strategy were often stronger than necessary for verifying the higher-level functions; some of that work could have been avoided although this stage was useful to familiarize ourselves with the code base, input-dependent loops, etc., and provided early, encouraging verification results.

Finally, this 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 be able to perform those steps intelligently and automatically with less or no user input.

Acknowledgments. We thank Ella Bonnimova for her help and feedback. We also thank Valentin Wüstholz, Alexander Summers, and Nikhil Swamy for their comments on drafts of this paper.
9. REFERENCES


