Lost in the Loader: The Many Faces of the Windows PE File Format

Dario Nisi
EURECOM
dario.nisi@eurecom.fr

Mariano Graziano
Cisco Talos
magrazia@cisco.com

Yanick Fratantonio
Cisco Talos
yfratant@cisco.com

Davide Balzarotti
EURECOM
davide.balzarotti@eurecom.fr

ABSTRACT

A known problem in the security industry is that programs that deal with executable file formats, such as OS loaders, reverse-engineering tools, and antivirus software, often have little discrepancies in the way they interpret an input file. These differences can be abused by attackers to evade detection or complicate reverse engineering, and are often found by researchers through a manual, trial-and-error process.

In this paper, we present the first systematic analysis and exploration of PE parsers. To this end, we developed a framework to easily capture the details on how different software parses, checks, and validates whether a file is compliant with a set of specifications. We then used this framework to create models for the loaders of three versions of Windows (XP, 7, and 10) and for several reverse-engineering and antivirus tools. Finally, we used this framework to automatically compare different models, generate new samples from a model, or validate an executable according to a chosen model. Our system also supports more complex tasks, such as "generating samples that would load on Windows 10 but not on Windows 7."

The results of our analysis have consequences on several aspects of system security. We show that popular analysis tools can be completely bypassed, that the information extracted by these analysis tools can be easily manipulated, and that it is trivial for malware authors to fingerprint and "target" only specific versions of an operating system in ways that are not obvious to someone analyzing the executable. But, more importantly, we show that there is not *one* correct way to parse PE files, and therefore that it is not sufficient for security tools to fix the many inconsistencies we found in our experiments. Instead, to tackle the problem at its roots, tools should allow the analyst to select *which* of the several loader models they should emulate.

CCS CONCEPTS

• **Security and privacy → Software reverse engineering**; **Malware and its mitigation**.

KEYWORDS

executable file formats, malware analysis, parser differentials

RAID '21, October 6–8, 2021, San Sebastian, Spain

ACM Reference Format:
Dario Nisi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. 2021. Lost in the Loader: The Many Faces of the Windows PE File Format. In *24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '21), October 6–8, 2021, San Sebastian, Spain.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3471621.3471848

1 INTRODUCTION

Over the past thirty years, malware authors have developed many techniques to bypass both static and dynamic analysis tools. The goal of the attackers is to either bypass or reduce the effectiveness of malware analysis tools while still producing samples that the target system would correctly execute.

For example, in a recent study conducted by Cozzi et al. [12], the authors discovered that malware authors often tamper with the executable file format to obtain binaries that are executed correctly on the target device but are discarded as malformed by the vast majority of the analysis tools (including disassemblers, debuggers, and common utilities to inspect the file headers). Along the same line, in 2017, Kim et al. [20] discovered a set of problems in the way AV products parse and validate signed PE files. Specifically, the authors noticed that if malicious files contain the Authenticode signature copied from a benign application, they are not analyzed. Even worse, many AV engines saved time by not even scanning signed binaries at all.

While these studies point out interesting discrepancies, we believe these findings are just the tip of the iceberg of a much deeper problem: while the inner structure of executable file formats is defined and generally well understood, the way these files are parsed and validated differs significantly among tools and operating systems and, surprisingly, also among different components of the same operating system [18].

Today, the security industry employs a completely automated malware collection, analysis, and classification process to handle the massive number of new samples discovered every day. This relies on a complex toolchain that combines many components to extract static features, collect the runtime behavior from malware analysis sandboxes, and compare each sample with information extracted from similar programs. Sadly, all the existing infrastructures rely on a subtle and often overlooked assumption, i.e., *that every single component should parse, understand, and validate the sample in the same way*. Ideally, the task of parsing the executable file format should be delegated to a common standard library used by all components. In practice, instead, every program implements its own parsing and

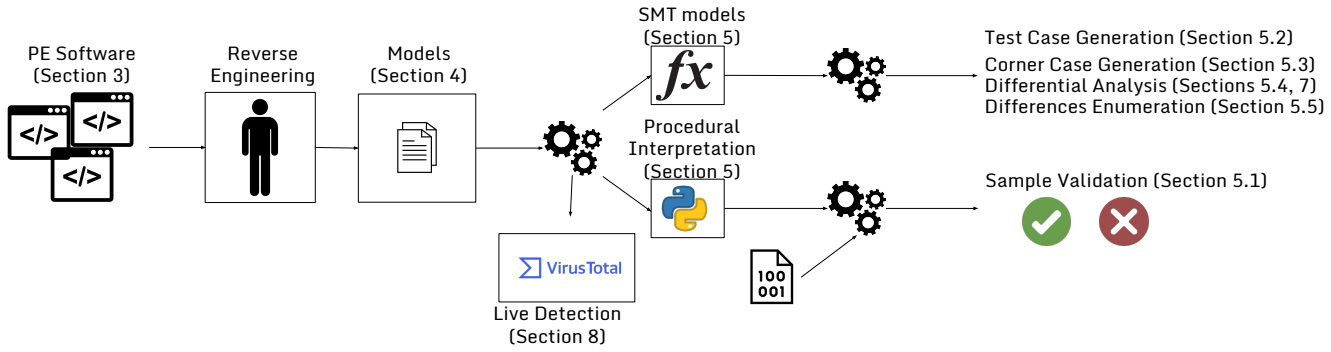

Figure 1: An overview of our analysis process.

validation routines, resulting in a multitude of strategies that often differ in many relevant details. Even worse, these techniques are not the same as those adopted by the operating system to decide whether a binary can be correctly loaded in memory and executed. On the one hand, these differences may result in samples that are erroneously discarded (because they are considered malformed by some static tools) or only partially analyzed. On the other hand, as measured by Ugarte-Pedrero et al. [36], it can result in the fact that a large number of damaged files are still assigned to dynamic analysis sandboxes — thus wasting a considerable amount of time and precious resources for security companies.

In the past, researchers have tried to look at this problem by partially documenting to what extent the structure of a PE file can be manipulated without compromising its functionality [3] or by collecting notes on some parts of the Windows loading process [35]. However, these studies followed a simple trial-and-error approach that failed to capture the scale and complexity of the problem. In fact, previous attempts often resorted to fuzz the file header fields to test whether the resulting file could still be executed in the system. However, this approach does not account for possible inter-relationships between fields (in which different parts of the file need consistent information) and therefore provided limited results.

The goal of this paper is to shed light on this complex problem by performing a comprehensive analysis of the factors that affect the parsing of the PE executable file format and on the fields that are read and used by different software and operating systems. An overview of our contributions is depicted in Figure 1. In particular, to perform our systematic exploration, we developed a new framework to precisely describe the steps commonly performed by OS loaders and file parsers. While this paper focuses on the PE file format, the framework is generic enough to support the description for other formats. We started our analysis by focusing on the OS loaders used in different versions of Microsoft Windows: Windows XP, Windows 7, and Windows 10. For each version we wrote a model that lists the checks and operations that are performed to determine 1) if the file is a valid PE and thus should be "loaded" in memory and 2) how the loading process extracts and parses information from the file. We also focused our attention on different categories of security-related software that deal with PE files, such as reverse-engineering tools and antivirus programs.

An analyst can use our models to perform several different tasks, such as sample validation, sample generation, corner case generation, differential analysis, and differences enumeration. Thanks to these fully automated techniques, our framework was able to *systematically* enumerate the discrepancies that exist between the Windows loaders and popular reverse engineering tools. Our findings have significant repercussions.

First, we show how popular analysis tools can be completely bypassed and how the information they extract can be easily manipulated "at will." We also ran a VirusTotal LiveHunt session and discovered that real-world malware is currently abusing some of these (previously unknown) discrepancies in-the-wild. These findings highlight how the research community lags behind malicious actors, thus making our systematic exploration of these aspects even more timely and important.

Second, we show that the differences in which the three versions of Windows parse PE files allow attackers to create targeted executables that would run on a specific version but would be discarded as malformed by others. This could be used, for example, to evade common malware analysis sandboxes, which often run Windows 7. However, by far, the most important consequence of the differences among OS loaders is that not only the PE standard fails to describe how an executable should be parsed and validated, but also that *a de-facto reference implementation does not even exist*. Instead, our experiments show that there are as many ways to interpret a PE file as there are versions of Windows, and therefore security tools should decide which model to use on a case-by-case basis. In other words, we show that since there is not a single correct way to parse PE files, fixing security tools is significantly more complex than just "fixing bugs." Instead, we believe that the only way to tackle this problem at its root is to offer the possibility to select *which* of the several loaders a tool should mimic, in order to adapt the validation and the extracted information to the system under analysis.

In the spirit of open science, we release the entire source code and datasets produced for this work at https://github.com/eurecom-s3/loaders_modeling, and we hope this will inspire a community-driven effort to refine our models and to extend them to different file formats.

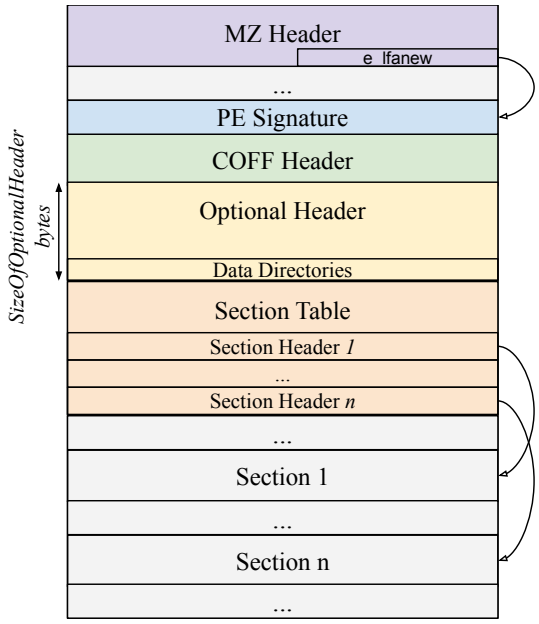

Figure 2: Structure of a PE Executable

2 THE PORTABLE EXECUTABLE FORMAT

Before diving into the *PE Format* anatomy, it is useful to introduce some terminology that we will use in the rest of the paper. We define the term "PE executable" (or simply "executable") to mean a file that follows the specifications of the *PE Format*, while we call "Process Image" (or simply "image") the representation in memory of the executable after it is loaded. In the paper, we also use the term "parser" in an informal way to refer to the general activity required to load the data contained in a PE file and map it to a set of predefined data structures. Finally, we call "validation" the process required to verify whether the information in a PE file satisfies a set of structural and logical constraints. As explained in Section 3, different classes of applications may parse and validate PE files in different ways and may take different actions when such constraints are not satisfied.

The PE format is the standard executable file format supported by the Windows operating system family [26]. Adopted by Microsoft since the release of Windows 3.1, this format underwent a series of revisions throughout the years that added support for new features. However, its core design remained unchanged and consists of a number of structured data types, commonly called "headers."

Figure 2 shows the structure of a *PE* executable. The first header at the beginning of the file is the *MZ Header*, originally used in the MS-DOS operating system and still in use today for backward compatibility. For Windows-specific executables, the *MZ Header* is only used for determining the offset at which the *COFF Header* starts. This second structure contains important information about the executable, including the architecture on which it is meant to run, whether it is a dynamic library or a standalone executable, and whether its image supports a randomized base address.

The *Optional Header*, which starts right after the *COFF Header*, provides more detailed information, including the preferred *ImageBase* (i.e., the virtual address of the first byte of the image in case no relocation is applied), the *SizeOfImage*, and the amount of memory to reserve for the stack and heap. The peculiarity of this header is that its size is not fixed but rather determined by the *SizeOfOptionaHeader* in the *COFF Header*. This design choice makes the *OptionalHeader* easy to extend in future revisions of the format specifications. Other fields of the *Optional Header* worth mentioning are the *Subsystem*, *MajorSubsystemVersion*, and *MinorSubsystemVersion*. The former indicates the Windows Subsystem required to run the executable (e.g., a program using the graphic user interface requires the Windows GUI Subsystem). The other two specify the minimum version of the Subsystem required.

The last portion of the *Optional Header* contains the *DataDirectory* table. A *DataDirectory* contains the relative virtual address (i.e., the offset from the base address the memory image; from now on *RVA*)) and the size of an additional data structure used for multiple purposes. Examples of *DataDirectories* are the *Import Table*, which declares the dynamic libraries and the functions that need to be loaded alongside the executable; the *Export Table*, containing the functions that the executable makes available for other programs to use; the *Relocation Table*, which provides a set of instructions on how to patch the executable in memory when it is not loaded at its preferred base address; and the *Certificate Table* that contains the digital signatures of the developer of the executable. The number of *DataDirectories* is not fixed and it is determined by a field in the *Optional Header*.

The *Section Table* starts at the end of the *Optional Header*. Each entry in this table defines a section, i.e., a contiguous memory region of the image, either uninitialized or populated with portions of the executable. Each section has its name and characteristics, such as the memory access permission level or whether the operating system can swap out its pages in case of a memory shortage. Sections logically organize the executable in homogeneous portions. For example, by convention, the *.text* section contains the code of the program, while the *.bss* contains the uninitialized data.

Constraints and Ambiguities. In addition to defining the structures of the headers, the *PE Format* specifications introduce "constraints" on their fields. With this term, we indicate a set of conditions that the fields must respect to be considered "acceptable." Trivial examples of constraints are that the first fields of the *MZ* and *COFF Headers* must contain their respective *magic numbers*: MZ and PE.

Other constraints are instead more subtle and complex. For example, each entry in the Section Table is expected to start at a multiple of *SectionAlignment* and populated with portions of the executable starting at an offset multiple of *FileAlignment* (both these are fields of the *Optional Header*). Moreover, they are expected to be sorted in ascending order by their starting virtual address, and adjacent entries in the table must be contiguous.

However, what the *PE Format* specifications fail to convey is what happens in case an executable does not meet such constraints. In other words, the specifications do not address the following questions "What happens if a section does not start at a multiple

of *Section Alignment*?" or "Can a PE executable whose sections are not sorted be considered valid nonetheless?"

Other ambiguities in the specifications concern the concepts of "default values" and "architecture-specific" features. The *Section-Alignment* field itself is an example of the first category. According to the specifications, the "default value" of this field is the page size in bytes of the architecture (e.g., 4K for Intel x86). Considering that sections are also used for specifying the memory access protection level of the corresponding memory regions, it is reasonable to assume that the value of the *Section Alignment* should be at least as big as the page size. In fact, a value of *SectionAlignment* smaller than the page size of the architecture could prevent the operating system from correctly enforce memory access permissions in adjacent sections.

OS loaders could handle this corner case in different ways. A strict loader could discard any executable with a value of *Section Alignment* smaller than the page size, while a more permissive one could still load the executable but without enforcing the access permissions of the sections. The important aspect is that the PE specifications do not provide any guidance about this implementation choice and provide no insights into handling this and similar other cases.

Finally, some relocation types described in the specifications fall within the "architecture-specific" features. For instance, the possible values of the *Base Relocation Type* enumerator are "meaningful" only in certain architectures. The concept of "meaningfulness," however, is not better elaborated in the specifications leaving, again, plenty of room for different implementation strategies. Should a programmer developing software that parses the *PE Format* consider relocation types that are "non-meaningful" for the targeted architecture as a violation of the specification (and interrupt the parsing of the file) or simply ignore them?

To summarize, the *PE Format* specifications do not clearly specify all the circumstances under which a file should or should not be considered a valid PE executable, nor do they provide unequivocal guidelines on how to handle corner cases. This, alongside the large amount of software that needs to deal with PE files, leads unavoidably to discrepancies in how PE executables are handled.

3 SOFTWARE HANDLING PE FILES

There are many classes of software that need to operate on PE files, for different reasons and with different objectives. In this section, we explore some of these classes, by grouping them according to their purposes and by discussing what are the basic operations that each group must perform to carry out its tasks.

3.1 Basic Operations on PE Executables

We identify three main operations that are performed by software that deals with the *PE Format*. Note that we call these "operations" and not "phases" because, as we found out in our work, they are usually interleaved and not implemented as self-contained, sequential stages in the software workflow.

Structural Checks. Operations of this type ensure that the file respects the basic structure of the *PE Format*. For instance, tests to verify the magic numbers or that the offset of a data structure points within the file boundaries are examples of structural checks.

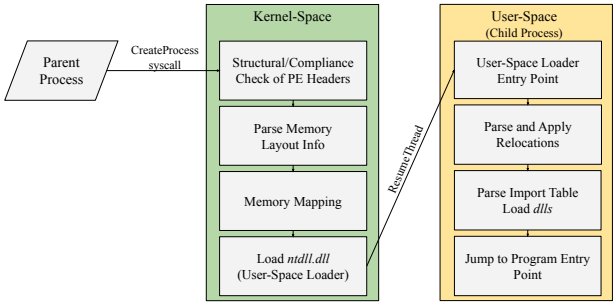

Figure 3: Windows Loading Process

Depending on the purpose of the software, these checks can be either strictly or loosely enforced. For example, some programs adopt a best effort approach and focus on gathering as much information as possible from the PE headers. For this reason, they might not abort the entire process if a structural requirement is not met, instead preferring to continue the process by focusing only on those parts of the headers that respect the PE structure. Other software may implement instead a more rigid structural validation, thus rejecting the files in which a structural check fails. Even within the same piece of software some structural checks can be enforced more strictly than others. This is because not all PE headers play the same role in all the software implementations of the format: Some may be essential, while others may be used only for optional features.

Compliance Checks. This type of operation ensures that the PE executables conform to both software- and architecture-specific constraints. For instance, the *Machine* field of the *COFF Header* specifies the CPU type on which the executable is meant to run. Operating systems may use this field to determine whether a program can be executed on the current CPU architecture. Compliance Checks are usually strictly enforced, and a violation of their constraints often results in rejecting the file.

As a rule of thumb, structural checks verify if a file is well formed, while compliance checks test whether the extracted information is valid and can be used to perform the task of the software. However, the boundary between *structural* and *compliance checks* is not always clear because of the ambiguities in the specifications of the *PE Format* that fails to declare the structure of a PE executable in a formal way. Despite this difficulty, we believe the distinction among the two types of checks can ease the discussions of the various aspects of our work.

Memory Mapping. These operations use the gathered information to prepare the *process* address space by creating a memory representation of the PE file suitable for execution. This image includes memory areas initialized with data extracted from different portions of the executable, memory areas mapped in the process address space but left uninitialized, as well as a description of the memory access permissions for each memory areas.

3.2 PE Software Landscape

Many classes of software need to parse PE files, and for this paper we focus on three main categories, which we selected because

they play a particularly important role in the security domain and because we believe they exemplify the different types of operations described above.

OS Loaders. Their objective, as their name suggests, is to load the image of the program in memory. In the case of the Windows operating system, the "PE loader" is not a well defined, self-contained component. As Figure 3 shows, there are two distinct parts that contribute to the process of loading PE executables in memory. The first one is embedded in the Windows Kernel and allocates the memory for the new process, as well as the kernel structures that identify it. This stage of the loading process also loads the *ntdll.dll* library in the new process. Once this first component finishes its tasks, the execution of the new process starts from a function in *ntdll.dll*, which initiates the second part of the loading process.

Both stages enforce *structural* and *compliance checks* on the PE headers and the structural checks are usually enforced strictly on all the headers. Once the loader has gathered the information it needs, it proceeds to allocate and populate the memory needed by the program. In other words, operating system loaders need to perform *all* the three basic operations described above. This, alongside the fact that they are implemented in two different components, makes this class of software the most complex and challenging to analyze.

Reverse-Engineering Tools. Programs in this class are used for software binary analysis. For this purpose, they need to gather information from the PE headers to recreate a memory representation similar to what an operating system loader may produce. This means that they are mostly interested in *memory mapping* and often perform *structural checks* with a best-effort approach. For example, they might rely on the debug information stored in the *Debug Header* if it is available, but fallback on other heuristics if such header is not present or is malformed. On the other hand, to be able to analyze as many executables as possible, this class of software performs very few *compliance checks*.

Antiviruses. Software in this class performs both *structural* and *compliance checks* on the information gathered from the PE headers of the executable. These constraints are meant to ensure that the executable provides the data needed for their format-specific signatures. However, they may perform very little *memory mapping* operations, usually only to enable signatures to convert *RVAs* to offsets into the original executable file (e.g., the *rva_to_offset* function in *yara* [43]).

4 CONSTRAINTS MODELING

As we discussed at the beginning of the paper, researchers have already documented several inconsistencies[2, 6] in the way different programs parse PE files over the past years. These previous attempts have focused on black-box approaches that try to construct anomalous files and observe whether a given OS (or analysis tool) would process them correctly. Altough automated black-box approaches, e.g., fuzzing, can find interesting differences, they either could only do that in simple cases or without the ability to list all possible differences exhaustively. Moreover, as it is the case for software testing, whenever such approaches are unable to find differences, it is not possible to conclude that two applications process PE files in an equivalent way.

For this paper, we are interested in finding the same flavor of behavioral differences, but we take a completely different approach: we opted for translating the parsing procedure of the various applications into formal models, which we can then query to explore the space of "acceptable inputs" *systematically*. Our observation is that both structural and compliance checks, as well as memory mapping operations, can be ultimately deconstructed into a set of formulas and constraints over the fields of the PE structures.

Our main goal is thus to develop models to describe those formulas and constraints in a way that can be used in an automated fashion to 1) compare different models, 2) verify whether a file is compliant with a given model, and 3) generate new PE files that satisfy the union or intersection of a set of models. The advantage of our approach with respect to previous efforts is that it can capture aspects of PE parsers in a comprehensive way, thus allowing us to exhaustively explore the search space in a structured way — something that other automated approaches (such as fuzzing) could not do.

4.1 Constraints Extraction

In our study, we model the checks performed on all the PE headers that are relevant to the loading process. These include the *MZ header*, the *COFF header*, the *OptionalHeader*, and the *Section Table*. We also modeled the entries in the *Data Directory* that play a role during loading, namely the *Import Directory*, the *Import Address Table*, the *Loader Configuration Directory*, and the *Base Relocation Directory*.

Our first objective is to extract the list of checks that a program performs when it loads a PE file. Several static analysis techniques exist that may help to automate this process. For instance, both symbolic execution and taint analysis can keep track of all the operations a program makes on its inputs and translate them into formulas. However, these approaches have severe limitations when applied to complex software. For instance, the Windows loader often stores and manipulates information in linked lists. This makes the relationship between a byte in the original PE file and a byte "tested in memory" very complex to describe and it often results in over-tainting large parts of unrelated memory. Moreover, existing tools (even after a considerable manual effort) could not scale to a typical OS loader's complexity. In fact, OS loaders are tightly connected to other low-level components of the operating system (such as the memory management and the code responsible for populating all process data structures in the kernel), and, as a result, the inability to isolate the loader itself from the rest of the OS code often results in a constant path explosion and constitutes a major problem for symbolic execution and taint analysis engines.

While it might be possible to prepare the target code manually in a way that could be analyzed by automated techniques, the effort would need to be repeated for each application. We attempted to perform such a task during our research, but we realized that it was faster and more advantageous to manually extract the constraints than to reverse the code to understand which paths were safe to prune for the symbolic execution engine. Moreover, in the context of analyzing a small number of very complex real-world software, this manual process, when assisted with an automated regression testing environment, is significantly less error-prone

than complex automatic approaches (e.g., symbolic execution) due to various shortcuts these approaches need to use when dealing with complexity.

We now provide more details about how this manual effort was conducted on the Windows loader, by far the most challenging software we modeled. This approach can be used to model other proprietary products with slight adjustments, while open-source software products — such as the other pieces of software we modeled — present fewer reverse-engineering challenges. We began by identifying the functions *MiCreateImageFileMap* in *ntoskrnl.exe* and *LdrpInitializeProcess* in *ntdll.dll*, which can be considered as the entry points for the kernel- and the user-space phase of the loading process, respectively. Using *IDA Pro*, and assisted by the debug symbols provided by Microsoft, we decompiled these two functions as well as the other routines that they invoke. Most of these routines return either a success or an error code. Our manual analysis consisted of finding all these exit conditions, tracing back their dependency to the input file, and encoding them using our modeling language.

Once all the functions of interest have been modeled, we tested the model to find bugs or missing constraints. To do this, we generated a number of test cases and ran them while monitoring the operating system with a kernel-debugger (we will explain the technique for generating test cases in the next section). This allowed us to gain confidence in the accuracy of our models.

4.2 Modeling Language

We designed a custom language tailored for describing and encoding the knowledge acquired with our manual analysis effort. The rationale for this choice is that very few tools have been developed over the years for systematizing logic constraints like the ones we want to model. The few available options are not tailored for our purposes and relying on such tools would introduce a significant overhead for the analyst, making the modeling process more time-consuming. For example, the *SMT-LIB* language [5] lacks support for modeling loops, while Dijkstra's Guarded Command Language [14] does not support structured types, which proved to be extremely useful for modeling the Windows loaders.

Instead, our language is designed for the purpose of modeling compliance checks on the PE Format (or any other similar format, like ELF). Moreover, models written in our language can be automatically translated in *SMT* queries to ensure that the constraints of the model are coherent and to generate test cases that "satisfy" the constraints of the model. We discuss several possible use cases in the next section.

From a high-level perspective, each model is a list of statements. Our language supports various types of statements, each of which aims at capturing (and making it easy to capture) specific traits of the file format we want to describe. We now document the various statement types and features of our language. The reader can find a toy example of a model written in our language and an excerpt of one of the models of the Windows loader in Appendices A and C, respectively. Moreover, the interested reader can review the full models at https://github.com/eurecom-s3/loaders_modeling.

Input Definition. The first type of statements allows us to create an *input symbol* of a given size in bytes that other statements can predicate on. Although our language supports multiple input definition statements, in our models we introduce only one symbolic input representing the *entire PE executable*. By doing so, we can treat the different components of the PE format as interconnected and interdependent entities, allowing our models to capture complex constraints involving fields of different headers.

Symbol Definition. The second type of statements allows us to introduce additional *symbols* (e.g., labels) and associate them with the result of operations on input or other symbols. The current version of our language supports arithmetic and bit-wise operations, as well as more complex operations commonly used to deal with the PE Format, e.g., *ALIGNUP* and *ALIGNDOWN* for respectively rounding up or down to a certain power of two.

Predicates. The third type of statements allows us to specify boolean comparisons between expressions of existing symbols, which evaluates to either *true* or *false*. In these statements, comparison and logic operators can be used, as well as more complex operators, e.g., *ISPOW2* to check if the operand is a power of two.

Terminal Predicates. Predicates in our language can be either *terminal* or *non-terminal*. A terminal predicate *must* be satisfied for an input file to be considered compliant with the model. Non-terminal predicates, instead, do not have such a requirement and can therefore be helpful when dealing with conditional predicates, which we discuss next.

Conditional Predicates. These allow encoding predicates of the form $P: \ A \Rightarrow B$, where A is a non-terminal predicate. In other words, *if* the predicate A is true, *then* the boolean predicate B *must* also be true for the overall predicate P to be satisfied.

Structured Types. As we mentioned in Section 2, the PE Format consists of many structures containing different fields. Our language implements a type system that makes models more readable by enabling to cast structured types on the original file and allowing the use of mnemonic names for each of the header fields. Types can be defined as C-like structures and can be used in all the statements discussed above.

Loops. The last construct supported by our language allows the encoding of *loops*. Loops are frequently used when parsing PE files, for instance to enforce that all entries in a list or array satisfy the same constraint. Once again, all the statements discussed above are supported within a loop.

5 USING MODELS

Once a human analyst has modeled the software constraints in our language, the models can then be automatically translated into a formal representation that can be used for various use cases. Moreover, we note that while the (manual) model extraction may theoretically contain imprecisions (we discuss in Section 6 how we experimentally validated them), all subsequent analysis steps have soundness guarantees.

5.1 Sample Validation

The first use case of our models is to determine whether a given PE executable meets the requirements of a specific loader. For this purpose, we rely on the *procedural interpretation* of the model. This

technique consists of interpreting the statements in the model sequentially, applying each of them to the executable under analysis.

Symbols defined via *symbol definition* statements are assigned concrete values according to the input file and predicates are evaluated to their boolean values. In case any of the *terminal predicates* is *false*, the validation process stops, and the executable is marked as non-conforming to the model. Conditional predicates are also evaluated to concrete values, but in their case the sample is rejected only if their precondition is true as well. If all terminal predicates evaluate to true, and thus no constraint is violated in the process, the executable is marked as conforming to the model.

This use of our models can be relevant as part of the dynamic malware analysis pipelines that are employed by all major security companies that offer malware detection and analysis products. In particular, it could be useful as a reliable pre-filtering stage that either selects the appropriate sandbox (based on the OS that can run the sample) or quickly discards malformed binaries that would not run successfully anyway.

5.2 Sample Generation

In this second use case, we describe how we can automatically generate concrete samples compliant with a given model. This is possible because models written in our language can be transformed into SMT decision problems over the *BitVector* theory. The input of the problem is a symbolic *BitVector* (of fixed size) representing the executable file. Each type of statement in our language can then be translated in an appropriate SMT problem, as follows.

Symbol Definition and Structured Types. Each *symbol* defined in a model is transformed into a symbolic value that can be processed by an SMT solver. Moreover, for each operation that puts a new symbol in relation to existing symbols, our tool generates appropriate predicates that encode their relationship. Structured types are handled in a similar fashion, by using the appropriate offsets in the symbolic input to convert aliases into concrete predicates.

Predicates. Each predicate in our language is transformed to a boolean formula usable by an SMT solver. These SMT predicates are then used to create appropriate constraints, depending on whether they are terminal or non-terminal.

Terminal Predicates. Terminal predicates *must* be satisfied for an input file to be compliant with a given model. Our tool performs the logic conjunction of all the terminal predicates to create the SMT problem's predicate that it then feeds to the solver.

Conditional Predicates. Conditional predicates *must* be satisfied if their precondition evaluates to true. To model them compatibly with an SMT solver, for a conditional predicate of the form $C \Rightarrow D$, we create the following SMT constraint: $\neg C \vee D$.

Handling Loops. SMT theories do not have an equivalent construct to our loop blocks. To address this problem, we handle loops in our models by using *loop unrolling*. In other words, each statement in the loop block is executed up to a fixed amount of times (defined in the loop statement).

Finally, we create a single constraint that encodes the logical conjunction of all the constraints mentioned above (i.e., if P_n is the n-th constraint in the model, we consider the constraint $\bigwedge_i P_i$). We then feed this single constraint to the SMT solver and ask it to produce a concrete input that satisfies all the predicates captured by our model. In the case of models that describe the compliance checks performed by a program on a PE executable, the SMT solutions are effectively PE headers that pass these checks. We use this observation to generate valid executables for the software we modeled.

Generating PE Files with Valid Code. A PE file generated by the SMT solver would pass all the checks but it would not execute any meaningful user-space code (because the loader does not check this part). However, this makes it harder to test whether a generated sample is valid and executes correctly. Therefore, we added a component to specify which code the generated PE file should execute and "plug" it into the generated files. We (successfully) tested our system with two examples, the first executing a single "exit(0)" syscall and the second printing "Hello World!". For more details, Appendix B shows a concrete example of the translation of a model into an SMT problem.

5.3 Corner Cases Generation

Our next use case focuses on generating a multitude of different test cases that *explore* the corner cases imposed by the PE loaders. To do so, we leverage *non-terminal predicates*, i.e., predicates that acts as preconditions for *conditional statements*. Since they do not need to be satisfied, we can consider them as *free variables* of the SMT problem.

Based on this observation, we automatically derive a number of SMT problems in the following way. Let S be the predicate of the SMT problem, and Q a non-terminal predicate of the model. Now consider the two following SMT problems: $S' \leftarrow S \wedge Q$ and $S'' \leftarrow S \wedge \neg Q$. By construction, even if S' and S'' are different from S, their solutions (if they exist) necessarily satisfy the original problem S too (because, by design, S is less restrictive). However, these solutions may differ substantially. In fact, in the first, the free constraint is asserted, which means that the *conditional statements* that have Q as precondition must be satisfied. The same is not true for the solutions to the second problem.

We can generalize this process to an arbitrary number n of predicates used as conditions by building new problems that either assert or negate each combination of them—thus resulting in at most 2^n different generated samples. In practice, the actual number of generated test cases can be lower because there is no guarantee that some combinations of the *free constraints* are not incompatible with each other, thus making the corresponding SMT problem unsolvable.

5.4 Differential Analysis

Another use case of our work is to combine the models of two different software and generate samples that either satisfy both or only one of the two.

For example, we can select two SMT problems built from distinct models (for instance, two versions of Windows or a version of Windows and an antivirus), whose predicates are S_1 and S_2, respectively. Our system can then generate a third SMT model with the following predicate: $S_{diff} : \quad S_1 \wedge \neg S_2$. If this SMT problem is satisfiable, its solutions are samples that pass all the compliance checks of the first software but do not satisfy at least one of the constraints in the

second model. On the other hand, under the assumption that the models correctly reflect the behavior of the software, if the problem is not satisfiable, it *guarantees* that no such samples exist.

We can also use this technique to prove that the two models are *equivalent*. In fact, if we cannot generate any sample that satisfies S_1 but not S_2 and neither any input that satisfies S_2 but not S_1, we can conclude that the two models are enforcing the same constraints.

5.5 Differences Enumeration

In our final use case, we can take the *Differential Analysis* technique one step further to find the *exact* constraints in the two models that make them behave differently.

The idea is to use an iterative process that starts with the differential analysis between the two models to compare. If the differential analysis problem has no solution, the process terminates. If, instead, a solution is found, we identify the predicates in the negated model that were false in the produced test case. We then add these predicates to the set of constraints negated at least once (from now on, N).

The process then continues with solving SMT problems of this form:

$$S^n_{diff} \leftarrow S_1 \wedge \neg S_2 \wedge S_{support}$$

in which $S_{support}$ represents the logic conjunction between a subset of the predicates in N. By construction, the solutions of this SMT problem (if any) meet the constraints in $S_{support}$ but violate at least one constraint in S_2. We then add the new violated constraints to N before repeating the process until we have used all the subsets of N (including N itself) in one iteration.

Note that this approach does not guarantee that *all* the differences between the two models are found when the process ends. To have this guarantee, one should build the $S_{support}$ terms as the logic conjunctions of all the subsets of the predicates in S_2. However, this requires a number of iterations that grows exponentially in the number of predicates in S_2, limiting its applications in practice.

On the other hand, we believe that our approach represents a good trade-off between scalability and precision. In fact, the number of iterations required is lower since it is exponential only in the number of the negated predicates. Moreover, the models that we compare are substantially similar, meaning that only a few of the constraints are not coherent between the models.

5.6 Implementation

We implemented a model analysis framework (available at https://github.com/eurecom-s3/loaders_modeling) that can perform all the operations described above. The tool consists of three different components. The first is the *Language Front End* responsible for parsing the input models and lifting them into an intermediate representation in the form of an abstract syntax tree. This component also handles the typing system by resolving the mnemonic fields of the structured types into offsets in the symbolic input.

The second part is a *Python Backend*, which performs the *sample validation* task by sequentially validating each statement in the intermediate representation on a sample provided as input.

Finally, the core of the framework is the *Z3 Backend* that implements the logic to translate models into SMT problems and solve these problems by using the z3 SMT solver [13] to generate samples. It also provides an interface to combine an arbitrary number of models together and to perform *Corner Cases Generation*, *Differential Analysis*, and *Differences Enumeration*.

6 MODELS EVALUATION

For our study, we modeled the loading process of three versions of Windows: Windows XP SP 3, Windows 7 SP 1, and Windows 10 (v. 1909). On average, each model contains 269 statements, of which 78 are terminal predicates.

Since all constraints were extracted by manual analysis, which may be affected by imprecisions, we evaluated how tightly these models capture the behavior of the target software components. To this end, we conducted two sets of experiments to assess whether the models are under-constrained (i.e., "too loose" in accepting invalid files) or over-constrained (i.e., "too strict" in rejecting valid files).

6.1 Assessing Under-Constrainedness

The objective of the first experiment is to verify that our models are not under-constrained with respect to the real OS. That is, if our model says "this file is a valid executable for a given OS," then the OS should be able to load and execute that file. Our approach consists of generating samples "at the boundary" of our models and attempting to execute them in the real OS: if a valid file according to our model does not run in the OS, our manual analysis missed important constraints (i.e., the model is under-constrained).

To generate these "extreme" samples, we used the *Corner Cases Generation* technique introduced in Section 5.3. By construction, all these samples are guaranteed to be valid according to the model. Moreover, since this technique recursively explores all the relevant free variables in the model, the generated samples can be seen as representatives of all models' corner cases. In details we generated 80 test cases from the model of Windows XP, 72 for Windows 7, and 20 for Windows 10. Each sample has been generated by combining the operating system model with the model responsible for adding the code of an "exit(0)" syscall at the entry point. This allowed us to infer whether the sample ran correctly by observing its return code (*%ERRORLEVEL%* according to the Windows terminology).

All the test cases generated by each model ran successfully under the respective operating system.

6.2 Assessing Over-Constrainedness

The goal of the second experiment is to verify that our models are not over-constrained compared to the real OS, i.e., that our models do not include erroneous constraints that were not present in the loader code. In other words, if the OS can load a given file, then the corresponding model should output that "this file is valid." To this end, we first built a dataset of 2543 real-world PE executables that we collected from the community-driven repository of the Chocolatey [8] Windows package manager. Note that, although these samples are very likely valid PE executables, they would not necessarily run under *all* the three Windows versions under analysis. We processed each file with the *Sample Validation* technique (discussed in Section 5.1) and we identified the samples that our models flagged as invalid: given the nature of the dataset we start with, these samples represent potential imprecision of the models.

Lastly, we verified whether these invalid files would actually run under the corresponding OS.

Out of the 2543 samples, our models predicted that 632, 261, and 86 were invalid according to our models of Windows XP, Windows 7, and Windows 10, respectively. To verify the accuracy of the prediction, we then executed each of these samples in a VM running the respective version of Windows OS: *all samples failed to run, precisely as predicted by our models*. We believe this is strong evidence that our models are not over-constrained. We also investigated the reasons behind this high rate of invalid PE samples. The most frequent problem is that these are PE files targeting more recent Windows versions using the *SubsystemVersion* fields. Other samples are invalid because they target other architectures than Intel x86, or because they are kernel modules that cannot run as standalone executables.

As an additional experiment to check that the robustness of our execution setup, we also ran all the "valid" samples and verified that they were all loaded properly in the respective OS. Note, however, that some of them could not be properly executed due to missing DLL dependencies, which is a problem that is not related to whether a given sample is compliant with the PE format.

7 DIFFERENTIAL ANALYSIS

In this section we present the results of the differential analyses we performed on our models. In particular, we present the discrepancies we found between the versions of the Windows loader we analyzed, as well as those between each Windows loader and three open-source tools commonly used in malware analysis, namely the ClamAV antivirus [9], the yara signature engine [42], and the reverse-engineering framework radare2 [28].

For the three versions of the Windows loaders and ClamAV, we compare the models of their compliance checks and their memory mappings. On the other hand, we compare only the models of the memory mapping operations performed by yara and radare2. This asymmetry is due to the latter two not performing any compliance checks when analyzing PE executables.

The analyses were performed by using the *Differences Enumeration* technique presented in Section 5.5. For the sake of completeness, each discrepancy we report has been manually validated by feeding the corresponding software with the test cases that the differential analysis generated.

7.1 Discrepancies among Versions of the Windows Loader

Interestingly, we found differences among all these three versions of the Windows loaders. For what concerns compliance checks, we found a number of discrepancies, discussed next. We note these represent *the exhaustive list* of discrepancies between the models we created for this paper and that all these were found automatically.

[W1] ImageBase = 0. Windows 7 considers as invalid any executable with a value of *ImageBase* equal to 0. We did not find the same behavior in either Windows XP or Windows 10.

[W2] SizeOfHeaders ≥ offset(COFFHeader) + 0x5d. For executables with *SectionAlignment* greater than the page size, both Windows 7 and Windows 10 expect the *SizeOfHeaders* to be greater than the offset in the file of the *COFFHeader* plus a constant. Windows XP, on the other hand, does not enforce this constraint.

Table 1: Source of discrepancies and in which differential analysis they were found

	Discrepancies				
	W1	**W2**	**W3**	**W4**	**W5**
XP vs 7	✓	✓			✓
XP vs 10		✓	✓	✓	
7 vs XP					
7 vs 10			✓	✓	
10 vs XP					
10 vs 7	✓				✓

[W3] Relocations for other architectures. Both Windows XP and Windows 7 handle architecture-specific relocation entries, even if they are not "meaningful" for the running system. On the other hand, Windows 10 only allows generic and Intel x86-specific relocations.

[W4] AddressOfEntryPoint < SizeOfHeaders. If the *AddressOfEntryPoint* of an executable lies within the first *SizeOfHeaders* bytes of the image, Windows 7 and Windows 10 discard the executable as invalid. The same does not happen under Windows XP.

[W5] SizeOfImage > endof(SectionTable). Windows 7 is the only version that does not load executables in which the offset of the last byte of the *SectionTable* in the file is greater than the value of *SizeOfImage*.

Table 1 provides a summary of the discrepancies found during the differential analyses. In particular, the ✓symbols in the table indicate whether a discrepancy was found during a differential analysis of two versions of the loader. For example, the ✓in first row (XP vs. 7), first column (W1) indicates that the first discrepancy (*ImageBase = 0*) was found while generating samples that comply with the model of Windows XP, that did not satisfy the model of Windows 7.

It is interesting to note how our analysis did not find any discrepancy based on the *OperatingSystemVersion* fields. In fact, according to the specifications, these fields should indicate the "version of the operating system" required to run the executable. However, in our investigation, we did not find evidence of any check performed on these fields, which can, therefore, assume any value. This is an example of the significant differences between the specifications of the PE Format and its software implementations (which are both maintained by Microsoft). On the other hand, the Windows loader checks that the *(Major|Minor)SubsytemVersion* fields are within a range that varies across the three versions of the loader we analyzed. Certain values of these fields also enable some version-specific features, such as the *Control Flow Guard*[24] extensions of the *Load Config Directory*, which the Windows 10 loader validates if the *SubsystemVersion* is greater than "6.3." Version-specific features represent one more cause of discrepancies in the behavior of the Windows loaders.

Our differential analysis on the memory mapping operations also highlighted a difference in how Windows 7 and Windows 10 map PE executables compared to Windows XP. The difference

affects the first region of the memory map, the one that contains the PE headers. Each version of Windows maps the PE headers in the process address space. However, if the *Section Alignment* of the executable is greater than the page size, Windows 7 and Windows 10 copy in this region only up to *SizeOfHeaders* bytes, while Windows XP copies up to 4KB from the executable file.

7.2 Compliance Checks Analysis of ClamAV

We now document the results of a differential analysis to produce test cases that comply with the constraints enforced by the operating system loaders, but violate those of the antivirus. For this experiment, we focus on ClamAV. The net result of each of our findings is that while these executables would run under the different Windows operating systems without problems, ClamAV would not be able to consider them as "fully valid PE files," and it would not be able to use signatures that rely on specifics of the PE format [10, 11].

It is also interesting to note how the differences reported by these differential analyses are completely different from those reported between the different versions of the OS loaders—showing once more how each developer interpreted in her own way the file specification. We now discuss the discrepancies we identified.

[C1] SizeOfOptionaHeader. As we explained in Section 2, this field is used to determine the length in bytes of the *OptionalHeader*. ClamAV expects its value to be greater or equal to the size of the structure defined by the PE Format. None of the Windows loaders we analyzed in our experiments enforce this constraint.

[C2] NumberOfSections. ClamAV expects at most 96 entries in the *SectionTable*. To the best of our knowledge this was the maximum number of sections in an executable supported by older versions of the Windows loader, while none of the versions of Windows we analyzed still enforces this constraint.

[C3] Section Virtual Address. For executables with *SectionAlignment* less than the page size of the architecture, the Windows loaders *do not* check that the starting addresses of the sections are multiple of this value. ClamAV, on the other hand, performs this check regardless of the value of the *SectionAlignment*.

7.3 Memory Mapping Analysis of ClamAV, radare2, and yara

We modeled the memory mapping operations of three popular software that deal with the PE format, namely ClamAV, radare2 [28], and yara [42]. Each of these software needs to provide some sort of memory mapping of PE executables. For ClamAV and yara, this memory mapping can be used to write malware signatures that rely on the content of the memory image. Radare2, on the other hand, provides the user a full-fledged memory representation of the PE executable to support her analysis and reverse engineering.

Under the hood, memory mapping operations are usually implemented by means of a primitive that maps *RVAs* in the memory image to offsets in the original file. We therefore extracted and modeled the implementation of this primitive in all the three software and performed a differential analysis between them and the three versions of Windows. Our system found discrepancies in the memory mapping operations of all the three software. In particular, we discovered that the nature of these differences was the same and it was due to incorrectly mapping PE executables that have a *SectionAlignment* lower than the page size. For these executables, all three versions of the Windows loader we examined copy the entire file *as is* in memory, starting at *ImageBase*, regardless of the entries in the section table (thus resulting in *RVAs* to be equivalent to file offsets). On the other hand, the three tools always rely on the section table to convert *RVAs* into file offsets.

Our differential analysis automatically produced test cases that leverage this key difference, by exploiting the fact that PE executables with a low value of *SectionAlignment* do not need a section table at all. Once again, the fact that all these three software make the same mistake shows that the PE specifications do not provide a clear and comprehensive set of guidelines for handling PE executables, nor they describe accurately the actual implementation of the Windows loader.

8 BYPASSING POPULAR ANALYSIS TOOLS

The discrepancies we discussed in the previous section have significant repercussions in the field of malware detection and analysis, as they undermine the trust we have in popular tools. This section discusses several examples of real executables that are not correctly supported by popular tools. It then presents the results of our investigations to determine 1) whether these discrepancies can be found among benign samples, and 2) whether there is evidence of malware samples that already exploit them as part of their attacks in-the-wild.

ClamAV. We modified the headers of real PE files to deceive ClamAV into not considering them as fully valid PE executables and bypassing signatures relying on specifics of the PE format. For instance, by simply adding empty sections at the end of the section table and updating the *NumberOfSections* field accordingly, the malware would still be valid for the Windows loader, but ClamAV would have issues using some PE-specific signatures. With minor adjustments, malware authors could also easily abuse the other discrepancies by, for example, lowering the value of *SectionAlignment* below the value of the page size, or by modifying the virtual address of one of the entries in the section table, so that the new value is not a multiple of the value of *SectionAlignment*. We created several proofs-of-concept that showcase these problems, and none of these attacks altered in any way the malware behavior.

Yara. We crafted a PE file that evades two key features commonly used in malware signatures: The resolution of imports and the pattern-matching applied at specific virtual addresses (implemented with the keyword *at*). The technique we used leverages the discrepancy in the memory mapping operation described in the previous section, and mainly revolves around creating a PE executable with a low value of *SectionAlignment*. To deceive the imports resolution, we placed the name of the imported DLL outside the memory ranges covered by the section table. The same can be done to mislead actual signatures that use the *at* keyword. For our example, we placed the entry point at an *RVA* that is not covered by the section table, but one could hide *any* portion of the code with the same technique.

Radare2. Reverse-engineering tools are not immune to mishandling the PE format either, and our experiments show that they

can be easily exploited to confuse both manual and automatic analysis based on these tools. As a proof-of-concept, we developed a technique to selectively hide portions of code from the radare2 framework. This was achieved by lowering the value of *SectionAlignment* and then adjusting the entries of the section table to minimize the number of bytes that radare2 maps in memory (by reducing the value of *SizeOfRawData* field in each entry of the section table).

Dynamic analysis pipelines. The showcased examples aim to evade static malware analysis, but dynamic malware analysis techniques can be evaded too. In fact, dynamic analysis pipelines rely on an instrumented version of an operating system to carry out their job. As we saw in the previous section, discrepancies among versions of the Windows loader exist, and no version can load *all* the programs that the others can. Most dynamic malware analysis pipelines run each sample only with one Windows version, which is often outdated and different than what end-users choose[44]. Without a way to statically identify the OS version to use, it is difficult to ensure that every file is correctly analyzed in the sandbox.

Measuring prevalence among goodware. Intuitively, one would not expect to find header "malformations" in benign software. In the vast majority of the cases, in fact, goodware is compiled by using mature and well-tested toolchains that are unlikely to produce headers that trigger different behaviors in different parsers. To test the validity of this assumption, we analyzed the dataset of goodware we used in Section 6.2, searching for samples that exhibit any of the causes discrepancies highlighted in Section 7. Out of the 2543 samples in the dataset, we found only three samples whose headers showcase any of the malformations. All three of them are resource-only images (i.e., PE files that do not contain any code, but only data) and had their *AddressOfEntryPoint* fields set to 0, matching the conditions of the *W4* discrepancy. According to the Windows APIs documentation [25], resource-only images undergo a different loading process, which does not perform many of the operations that the regular process does, ultimately enforcing fewer constraints on the PE headers. Thus, there is no evidence of prevalence among goodware of conditions that can lead to discrepancies in the PE loading process. In other words, in none of the samples in our dataset, could we find PE headers that would trigger any of the discrepancies reported in this paper. This suggests that, if such peculiar headers were to be found, they most likely would have not been produced accidentally.

Evidence of usage in-the-wild. At last, we wanted to investigate whether real-world malware is already exploiting the discrepancies we found. To this end, we run a LiveHunt campaign on VirusTotal [38] from Oct. 7, 2020, to Oct. 19, 2020. The LiveHunt service allows researchers to scan all the samples received by the VT platform with custom signatures, written in yara, in real-time. For our campaign, we wrote yara rules that match the discrepancies we found in the compliance checks reported in the last section, with the only exception of the one concerning architecture-specific relocations.[1] We also wrote one additional rule that matches samples that exhibit a value of *SectionAlignment* lower than the page size. As we saw throughout the paper, this is the precondition for all the discrepancies in the memory mapping operations.

[1]The PE module of yara does not provide APIs to access the relocation table, and parsing it using the yara language is not possible as it lacks support for a loop construct in which the number of iterations is not determined before the loop starts, which is necessary to parse the table.

Table 2: #Samples reported for each discrepancy

		Windows Loaders				Windows vs. ClamAV		
	Align	**W1**	**W2**	**W4**	**W5**	**C1**	**C2**	**C3**
# Samples	301	37	43	27	1	15	77	1
[0, 5) Detect.	59	14	3	15	0	0	1	0
[5, 10) Detect.	36	1	3	0	0	0	1	0
[10, 20) Detect.	5	4	2	1	0	1	0	0
≥ 20 Detect.	201	18	35	11	1	14	75	1

Our LiveHunt identified a total of 467 samples. Table 2 presents a breakdown of the samples grouped by which yara rule matched and by the number of AV detections. 73% of the samples were marked as malicious by 20 AVs or more, with an average of 36.6 AV detections (out of 74) per sample.

Despite evidence that the presence of discrepancies is rare in benign samples (as discussed earlier), it is challenging to determine that, with certainty, these LiveHunt samples are intentionally abusing them. Nonetheless, there are some cases for which there is indeed relevant evidence. For example, we found 77 samples with more than 96 sections, which would interfere with ClamAV's scanning process. Intrigued by the (relatively) high number of encounters, we manually analyzed some of these malware samples and found that they often include *exactly* 97 sections: this is unlikely a coincidence as this is the minimum number required to trigger the constraint in ClamAV. The relatively large number of samples using this trick suggests, in our opinion, that malware authors are actively employing this as an evasion technique. However, we do not know whether these tricks are used to specifically target ClamAV or whether they are additionally targeting other antiviruses, which may be affected by similar problems.

As another noteworthy example, VirusTotal reported one sample exhibiting the discrepancy involving the end of the section table (i.e., rule W5). It appears that this malware purposely crafted it to escape static malware analysis. In fact, on top of its header's peculiar structure, the sample also showcases several anti-analysis techniques, including anti-disassembly tricks and runtime library loading.

Last, most of the identified samples exhibit a value of section alignment lower than the page size. We believe it is likely that the samples' authors adjusted this value on purpose. In fact, the default value of section alignment for all the Windows toolchains we tested is precisely the page size, which is confirmed by very few encounters in regular PE executables.

To properly understand the results presented so far, it is important to contextualize these numbers with respect to the big picture. In fact, on an average week, VirusTotal processes around 3 million submissions of Windows executables [39], which suggests that the exploitation of the discrepancies is a relatively niche phenomenon at the moment. However, we believe that evidence of in-the-wild use of these discrepancies is concerning. To the best of our knowledge, we are the first to report these discrepancies publicly, and we

hope our work will help the community deal with this problem in a timely manner.

9 DISCUSSION

The results presented in this paper prove that different software handle the PE format in different fashions, leading to incongruities in both *compliance checks* and *memory mappings*. Our models show that there are as many ways to interpret a PE file as there are versions of Windows. In Section 8 we showed how these discrepancies allows attackers to bypass popular analysis tools and pipelines, and to create targeted executables that would run on a certain version but would be discarded as malformed by the loaders of other versions. We reported the bugs we identified to the developers of the tools; ClamAV developers have already acknowledged the problem and they are working on fixes. We believe, however, that it would be significantly more complicate to "properly" address the various aspects discussed in this paper.

There is not a single reference (or even a correct) implementation. We believe that security tools should allow the user to decide which model to use on a case-by-case basis. We note that this goes beyond the relatively well-known "the analyst should be able to select which type of Virtual Machine / Windows version to use for dynamic analysis " as the problem affects all static reversing tools as well. As we have shown in this paper, popular reversing tools can be completely bypassed and they can be fooled to return misleading and/or inaccurate data to the analysis client (e.g., a human analyst or a processing block in an automated pipeline)

So far, *no static reversing tool (including the popular commercial options) has even the "concept" of letting the user selecting which loader to emulate.* Over the last decade, instead, they have all tried a best-effort approach to implement a *single* loading process that attempts to be flexible and catch the various differences. This paper wants to sound the alarm bell that this long-time effort is bound to failure as *there is not a single "correct" implementation*, but there are as many as the OS versions. Other than fixing bugs to patch the discrepancies, we argue that the correct approach to eradicate this problem is to continue our effort to document the parsing models adopted by different software implementations and encode this knowledge in formal models that can be included in other tools and selected by the user.

On the need of formal specifications and reference implementations. Until now, developers of security tools had to implement their own PE parsers as the constraints and operations performed by the Windows loaders were not documented. At first glance, parsing the PE Format may seem a simple task, on top of which a large body of research and commercial tools are built. As our experiments show, however, there are many corner cases that are not sufficiently described in the PE specification, and different tools and operating system handle them in very different ways. This is a systemic issue that represents a concrete threat especially in adversarial fields like malware analysis, in which every wrong assumption may open up new avenues for evading detection mechanisms.

We argue that such problem should be tackled at its roots. One of these is the lack of a formal specification of the format that defines what a well-formed PE executable looks like. The ambiguities in the current specifications have ultimately lead to incongruities in how PE executables are handled. A possible solution to this problem could be to systematize the PE Format by means of formal methods. This would not only ease the development of new tools and loaders, but it would also provide reliable ways to discover discrepancies in their implementations.

One other way to avoid discrepancies among the large number of software dealing with the PE format could be to have a well-documented, publicly available reference implementation of the Windows loader, to which the new versions of the loader comply. This would ease the task of writing new reverse engineering tools and antiviruses, since they would not require developers to guess or to manually reverse engineer the Windows operating system internals.

While we believe this to be the best solution for the future, our work provides a solution to deal with both past and present versions of MS Windows. All our models and the tools required to use them for validation, test case generation, and comparison among tools are available at https://github.com/eurecom-s3/loaders_modeling. Thanks to our effort, PE parsing libraries and tools (such as pefile [15] and pev [27]) can provide different options to their users for choosing to interpret and/or validate a file according to one model or another.

10 RELATED WORK

Program loaders are core components of an operating system and, as such, have been widely studied by the research community. We therefore organize the discussion of previous research in this field along four categories.

Edge cases. Researchers have shown the limitations of static analysis approaches in parsing binary formats like ELF [18] and PE [29, 33], by abusing relocations to hide malicious code from static analysis tools. Ge et al. [16] used relocations to alter the memory permissions with severe security implications. Other researchers showed how a single byte can break several ELF parsers and still execute on the target machine [37], and how to use ELF metadata to backdoor a `setuid` application [31]. For what concerns the PE ecosystem in particular, Albertini [2] created PE executables that executes different instruction on different Windows versions, by leveraging discrepancies in the loaders implementations of these operating systems. Albertini also developed a collection of proof-of-concept binaries that showcase exotic features of the PE file format [1]. Previous works, like those by Vuksan et al. [40] and Huang [17], acknowledged that PE specifications leave space for implementation choices and documented a series of "malformed" (yet valid) PE header layouts. All these works only scratched the surface of the problem of discrepancies in the PE ecosystem. In fact, all studies discussed only anecdotal examples and the authors never attempted to generalize nor to propose a comprehensive approach to eradicate the problem.

Differential Parsing. The problem of differential parsing arises when different implementations of parsers for the same format produce discordant results when fed with the same input. Researchers presented several attacks based on differential parsing. For instance, Kaminsky et al. [19] demonstrated an attack against the X.509 infrastructure. Other attacks allowed privilege escalation on mobile

systems, like the infamous Android "master key" attack [30] or the more recent iOS 0day for the plist parsers [32]. Bratus et al. showed how to create a file that the Linux kernel- and user-space loaders parse differently [6]. These works show how researchers found inconsistencies among implementations of parsers of the same format. However, all these cases had been discovered manually without any guarantee of completeness. In this paper we instead propose a framework to automatically find all the incongruities among two tools. While we only presented our techniques applied to the PE ecosystem, we also created models for software parsing the ELF format and we are confident that our approach can be expanded to other formats as well.

Evasion. Several studies have focused on detecting malware evasive behaviors [21, 23, 41] or have attempted to measure their prevalence, such as in the case of stalling code [4, 22]. The work presented in this paper fits in the line of research that identified in the mishandling of executable file formats a major avenue for malware analysis evasion. Previous works have presented single instances of this problem. For instance, Petsios et al. [34] showed two cases in which ClamAV failed to parse malicious ELF files that properly run on the operating system. Similarly, Kim et al. [20] showed that many AV engines do not scan signed PE files and do not accurately validate the Authenticode signature that can be copied from benign applications. Again, this shows how evasion attacks are possible when AV engines handle PE files differently than the OS loader.

Automatic Modeling of Protocol Stacks. Brumley et al [7] presented a taint tracking-based automatic approach to create SMT models of the behavior of web servers handling HTTP requests. They used these models for differential analysis with a technique similar to the one we presented in our work. However, due to the intrinsic limitations of taint tracking, the models they produced cannot be complete.

11 CONCLUSION

In this paper we presented the first systematic and comprehensive exploration on discrepancies on how software deals with Windows PE file format. We have designed a custom language that allows us to build powerful and detailed models on the inner workings of different categories of software, ranging from OS loaders to reverse-engineering and antivirus analysis tools. We show how these models are powerful, as they allow researches to perform a wide range of tasks, such as sample validation, sample generation, corner case generation, differential analysis, and differences enumeration.

The results of our experiments show that 1) popular analysis tools can be easily evaded, 2) it is trivial for malware to fingerprint and "target" only specific versions of Windows, and 3) there is not *one* correct way to parse PE files, and that security tools should not only fix the many inconsistencies we found, but, to tackle the problem and its roots, should also allow an analyst to select *which* of the several loaders should be emulated. Moreover, we have found evidence that real-world malware is already abusing some of these discrepancies. We note that the power of our models is not limited to the results we obtained, but that it was possible to obtain these results in an automated and systematic way. By releasing our framework and datasets, we hope this work will inspire a community-driven effort to refine our models and to enable analysis software to deal with this systemic problem.

ACKNOWLEDGEMENTS

This project was partially supported by the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (grant agreement No 771844 BitCrumbs), and by CISCO Grant No 1377523. We would like to thank the anonymous reviewers for their constructive and insightful feedback. We would also like to thank Andrew Williams and Micah Snyder for their feedback and help with this project. Finally, in addition to thank, as always, Betty Sebright, we would also like to acknowledge Savino "amico caro" Dambra, who taught us what true friends are, and Fabio "we should attack more" Pagani, who taught us that one (or, at least, he) cannot win all games one plays.

REFERENCES

[1] A. Albertini. [n.d.]. Corkami PE files corpus. https://github.com/corkami/pocs/tree/master/PE.

[2] A. Albertini. 2013. Making a Multi-Windows PE. *POC or GTFO* 0x01 (2013).

[3] Alexander Sotirov. [n.d.]. TinyPE. http://www.phreedom.org/research/tinype/.

[4] B. Baker, A. Chiu. 2015. Threat Spotlight: Rombertik – Gazing Past the Smoke, Mirrors, and Trapdoors. https://blogs.cisco.com/security/talos/rombertik.

[5] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. 2010. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*, Vol. 13. 14.

[6] S Bratus and J Bangert. 2013. ELFs are dorky, elves are cool. *POC or GTFO* 0x00 (2013).

[7] David Brumley, Juan Caballero, Zhenkai Liang, James Newsome, and Dawn Song. 2007. Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation.. In *USENIX Security Symposium*. 15.

[8] Chocolatey. [n.d.]. Chocolatey - The Package Manager for Windows. https://chocolatey.org/

[9] Cisco. [n.d.]. ClamAV. https://www.clamav.net/

[10] Cisco. [n.d.]. ClamAV - Bytecode Signatures. https://www.clamav.net/documents/bytecode-signatures

[11] Cisco. [n.d.]. ClamAV - File hash signatures. https://www.clamav.net/documents/file-hash-signatures

[12] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. 2018. Understanding Linux Malware. In *IEEE Symposium on Security & Privacy* (San Francisco, CA). IEEE Computer Society.

[13] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the International Conference on the Theory and Practice of Software, International Conference on Tools and Algorithms for the Construction and Analysis of Systems (ETAPS/TACAS)*.

[14] Edsger W Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 8 (1975), 453–457.

[15] erocarrera. [n.d.]. pefile. https://github.com/erocarrera/pefile

[16] Xinyang Ge, Mathias Payer, and Trent Jaeger. 2017. An Evil Copy: How the Loader Betrays You.. In *NDSS*.

[17] Yinrong Huang. 2006. *Vulnerabilities in Portable Executable (PE) File Format For Win32 Architecture*. Technical Report. TR, Exurity Inc., Canada.

[18] J. Bangert, R. Shapiro, S. Bratus. 2013. Weird Machines and revisiting Trusting Trust for binary toolchains. http://www.cs.dartmouth.edu/~sergey/trust/30c3-chain-of-trust.pdf.

[19] Dan Kaminsky, Meredith L Patterson, and Len Sassaman. 2010. PKI layer cake: New collision attacks against the global X. 509 infrastructure. In *International Conference on Financial Cryptography and Data Security*. Springer, 289–303.

[20] Doowon Kim, Bum Jun Kwon, and Tudor Dumitraş. 2017. Certified malware: Measuring breaches of trust in the windows code-signing pki. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1435–1448.

[21] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. 2014. Barecloud: bare-metal analysis-based evasive malware detection. In *23rd USENIX Security Symposium (USENIX Security 14)*. 287–301.

[22] Clemens Kolbitsch, Engin Kirda, and Christopher Kruegel. 2011. The power of procrastination: detection and mitigation of execution-stalling malicious code. In *Proceedings of the 18th ACM conference on Computer and communications security*. 285–296.

[23] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. 2011. Detecting environment-sensitive malware. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 338–357.

[24] Microsoft. 2018. Control Flow Guard. https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard.

[25] Microsoft. 2018. LoadLibraryExA – Windows API. https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibraryexa.

[26] pe format [n.d.]. PE Format. https://docs.microsoft.com/en-gb/windows/win32/debug/pe-format

[27] pev [n.d.]. pev - User manual. http://pev.sourceforge.net/doc/manual/en_us/

[28] radare2 [n.d.]. radare2, a portable reversing framework. http://www.radare.org/.

[29] roy g biv / defjam. [n.d.]. Virtual Code Windows 7 update. https://github.com/darkspik3/Valhalla-ezines/blob/master/Valhalla%20%233/articles/VCODE2.TXT.

[30] saurik. 2013. Exploit (& Fix) Android Master Key. http://www.saurik.com/id/17.

[31] Rebecca Shapiro, Sergey Bratus, and Sean W. Smith. 2013. "Weird Machines" in ELF: A Spotlight on the Underappreciated Metadata. In *7th USENIX Workshop on Offensive Technologies (WOOT 13)*. USENIX Association, Washington, D.C. https://www.usenix.org/conference/woot13/workshop-program/presentation/shapiro

[32] Siguza. 2020. Psychic Paper. https://siguza.github.io/psychicpaper/.

[33] skape. 2006. Locreate: An Anagram for Relocate. http://www.uninformed.org/?v=6&a=3&t=txt.

[34] T. Petsios, A. Tang, S. J. Stolfo, A. D. Keromytis, S. Jana. 2017. NEZHA: Efficient Domain-independent Differential Testing. In *Proceedings of the 38th IEEE Symposium on Security & Privacy*. San Jose, CA.

[35] Todd Cullum. 2017. Portable Executable File Corruption Preventing Malware From Running. https://toddcullumresearch.com/2017/07/16/portable-executable-file-corruption/.

[36] Xabier Ugarte-Pedrero, Mariano Graziano, and Davide Balzarotti. 2019. A Close Look at a Daily Dataset of Malware Samples. *ACM Transactions on Privacy and Security (TOPS)* 22, 1, Article 6 (January 2019), 30 pages. https://doi.org/10.1145/3291061

[37] ulexec. 2019. ELF Crafting Advance Anti-Analysis techniques for the Linux Platform. https://github.com/radareorg/r2con2019/blob/master/talks/elf_crafting/ELF_Crafting_ulexec.pdf.

[38] virustotal [n.d.]. VirusTotal. https://www.virustotal.com/.

[39] VirusTotal. 2021. File statistics during last 7 days. https://www.virustotal.com/en/statistics/.

[40] Mario Vuksan and Tomislav Pericin. 2011. Constant insecurity: Things you didn't know about portable executable file format. In *BlackHat*.

[41] Zhaoyan Xu, Jialong Zhang, Guofei Gu, and Zhiqiang Lin. 2014. Goldeneye: Efficiently and effectively unveiling malware's targeted environment. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 22–45.

[42] yara [n.d.]. VirtusTotal - yara in a nutshell. https://github.com/VirusTotal/yara

[43] yara pe [n.d.]. PE module — yara 4.0.2 documentation. https://yara.readthedocs.io/en/stable/modules/pe.html

[44] Akira Yokoyama, Kou Ishii, Rui Tanabe, Yinmin Papa, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, Daisuke Inoue, Michael Brengel, Michael Backes, et al. 2016. SandPrint: fingerprinting malware sandboxes to provide intelligence for sandbox evasion. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 165–187.

APPENDICES

A Example of Constraints Model

```
1  INPUT foo 4
2  bar <- ADD foo 1
3  V1: ULE bar 10 term
4  V2: UGE foo 4
5  V3(V2): UGE foo 7 term
```

Listing 1: Example of a model written in our language.

Listing 1 provides a concrete example of a model written in our language. We now discuss the semantics of this model, and we will use this example as reference when explaining how other parts of our framework work.

At Line 1, the model specifies the existence of a 4-byte long input, named `foo`. At Line 2, a new symbol is introduced, `bar`, and the model specifies that it is *defined* as `foo + 1`. Note that multi bytes are parsed to integers as big-endian. Thus, adding 1 to a 4-byte field means adding 1 to its 4th byte, with carry. At Line 3, the model defines a boolean predicate, `V1`. The boolean predicate `V1` evaluates to true if and only if `bar` is less or equal than 10 (the "U" in ULE indicates it is an unsigned comparison). Moreover, the `term` keyword specifies that `V1` is a *terminal predicate*. This implies that, for an input file to be considered compliant, `bar`'s value *must* be less or equal than 10. At Line 4, the model defines another boolean predicate, `V2`. This predicate evaluates to true if and only if `foo` is greater or equal than 4 (once again, the comparison is unsigned). Note that, differently from `V1`, `V2` is *not* a terminal predicate. This means that, per se, whether `V2` evaluates to true is not a necessary condition for a given input file to be considered valid. The truth value of `V2`, however, is relevant for the conditional predicate `V3` specified at Line 5. The semantics of line 5 is the following: if `V2` evaluates to false, then `V3` evaluates to true, independently from the input value; if, however, `V2` evaluates to true, the specified predicate (i.e., *foo greater or equal than 7*) *must* also evaluate to true for `V3` to be satisfied. In addition to that, note that `V3` is a terminal predicate, which indicates that it *must* evaluate to true for an input file to satisfy the model. The net effect of this model is to constrain the value of the input in the ranges [0, 3] and [7, 9].

B Example of Translation in SMT problem

For sake of clarity, we now discuss how the example model discussed in the previous section in Listing 1 is translated into an SMT problem.

The symbol definition at line 2 introduces the following formula:

$$bar \leftarrow foo + 1$$

Line 3 introduces the following predicate:

$$P_1 : \quad foo + 1 \leq 10$$

Line 4 is translated into the following predicate:

$$Q_1 : \quad foo \geq 4$$

The conditional predicate is instead translated as

$$P_2 : \quad Q_1 \Rightarrow (foo \geq 7)$$

The final formula of the SMT problem is then computed as the logic conjunction of all terminal predicates, in this case P_1 and P_2:

$$F : \quad (foo + 1 \leq 10) \wedge (\neg(foo \geq 4) \vee (foo \geq 7))$$

Thus, the SMT solver will be tasked to find an foo such that the final constraint F is satisfied. In this case, the constraint *is* satisfiable, and the SMT solver would return an integer value in the ranges [0, 3] and [7, 9].

C Excerpts from the Models of the Windows Loader

```
1  ### Relocations
2  P: relocDir <- optHdr.DataDirectory[40, 8] as
       _IMAGE_DATA_DIRECTORY
3  P: relocVA <- relocDir.VirtualAddress
4  P: relocSize <- relocDir.Size
5
6  #### From ntdll!LdrRelocateImageWithBias:45,48
7  V6: AND (UGE optHdr.NumberOfRvaAndSizes 6) AND (NEq
       relocVA 0) (NEq relocSize 0)
8
9  P: tmpSize <- relocSize
10 P(V6): loopStart <- relocVA
11 L1(V6): relocBlockAddr <- VLOOP(loopStart, nextBlockAddr,
        V99, 10)
12     P: relocBlock <- HEADER[relocBlockAddr, 8]
13     P: blockSize <- relocBlock[4, 4]
14     P: blockPage <- relocBlock[0, 4]
15
16     P: firstEntryAddr <- ADD relocBlockAddr 8
17     P: nEntry <- SHR (SUB blockSize 8) 1
18     P: tmpSize <- SUB tmpSize blockSize
19     P: nextBlockAddr <- ADD relocBlockAddr blockSize
20
21     V99: NEq tmpSize 0
22
23     P: tmpEntry <- INT 0 4
24     V96: UGE nEntry 1
25     L2(V96): entryAddr <- VLOOP(firstEntryAddr, nextBAddr
           , V98, 10)
26         P: entry <- HEADER[entryAddr, 2]
27         P: tmpEntry <- ADD tmpEntry 1
28         P: nextBAddr <- ADD entryAddr 2
29
30         P: relocType <- SHR BITAND entry[1] 0xf0 4
31         P: relocAddr <- BITAND entry 0xfff
32
33         V11: EQ (BITAND (SHL one (SHR entry 12)) 0x3a0) 0
               term
34         V12: OR EQ relocType 10 ULE relocType 4 term
35         V13: ULT ADD blockPage relocAddr imageEnd term
36
37         ## RelocType 4 uses two entries instead of 1
38         V14: Eq relocType 4
39         P(V14): tmpEntry <- ADD tmpEntry 1
40         P(V14): nextBAddr <- ADD nextBAddr 2
41
42         ## From ntdll!LdrRelocateImageWithBias:47
43         ### Checks that the RtlImageNtHeader is still
               valid
44         ### Meaning that the targeted addre can't be 0x0
               or 0x1...
45         V15: OR (EQ relocType 0) AND (NEQ relocAddr 0) (
               NEQ relocAddr 1) term
46         ### Can't overlap with e_lfanew (0x3f-0x40)
47         V16: OR (EQ relocType 0) OR (ULT relocAddr 0x3f)
               (UGT relocAddr 0x40) term
48         ### Cant' overlap with PE magic (e_lfanew-
               e_lfanew+4)
```

```
49          V17: OR (EQ relocType 0) OR (ULT relocAddr HEADER
                 .e_lfanew) (UGT relocAddr ADD HEADER.
                 e_lfanew 4) term
50
51          V98: ULT tmpEntry nEntry
52      END L2
53  END L1
```

Listing 2: Excerpt of the model of the loader of Windows 10 handling relocations

Listing 2 shows the portion of the model of the loader of Windows 10 that handles the *Base Relocation* data directory.

Lines 2 to 4 parse the *RVA* and size of the relocation table. Line 7 introduces the boolean predicate `V6`, which evaluates to `true` when the relocation directory *RVA* and size are valid. All the following statements have `V6` as a precondition since they only make sense if the executable has a relocation table.

The logic that models the relocation table's content is embedded in the loop `L1` introduced at line 11 by the `VLOOP` operand. At each iteration of the loop, the variable `relocBlockAddr` contains the offset of the current relocation block in the file, starting from the value stored in `loopStart`. The boolean predicate `V99` (defined at line 21) is evaluated at the end of each loop cycle to determine whether the loop must continue. If that is the case, `relocBlockAddr` will be updated with the value stored in the variable `nextBlockAddr`. The last parameter of the `VLOOP` operand is the unroll count that indicates the maximum number of times the SMT solver must consider the statements in the loop.

Each relocation block is followed by a number of relocation entries which depends on the block's size. The loop `L2` handles each relocation entry for the current block. For the sake of brevity, we will not describe the parameters of the `L2` loop, as they are similar to the one of `L1`.

The terminal predicate at line 34 determines the types of relocations that the loader supports. For Windows 10, this predicate evaluates to true if the relocation type is either 10, or less or equal to 4.