DEFEATING POLYMORPHISM: BEYOND EMULATION

Adrian E. Stepan
Microsoft Corp., One Microsoft Way, Redmond, WA 90852, USA

Tel +1 425 706 9498 • Fax +1 425 936 7329

ABSTRACT

The most common method of detecting malware relies on signatures extracted from the malware body. Attempting to defeat this method and evade detection, malware writers have resorted to code obfuscation techniques, thus creating polymorphic viruses.

There are several well-known methods of decrypting polymorphic viruses, such as emulation, cryptanalysis (x-ray) and dedicated decryption routines. Each of these methods has some limitations: x-ray can handle only simple decryptions; dedicated routines require significant development effort and neither scales well with the number of detected viruses. Emulation doesn't have these weaknesses, but emulating code is significantly slower than executing it on a real CPU. Therefore a very complex polymorphic virus would take an unreasonable length of time to emulate until it is decrypted.

This paper proposes a new method of dealing with polymorphic malware. The method relies on disassembling the analysed code dynamically and performing just-in-time compilation targeted for the host CPU. The code obtained as a result can be executed safely on the host CPU, with little degradation in execution speed, compared to the original code. This provides the same flexibility as emulation but performance, in terms of speed, is dramatically improved. Additionally, the method could be used for other purposes, such as generic unpacking of packed executables, and behaviour-based analysis of complex code.

1. DETECTING MALWARE, AN ONGOING BATTLE

Malware has a long history of evolution. During the past two decades, malware has evolved with regard to replication and spreading mechanisms, as well as techniques used to prevent analysis and/or detection. Such techniques include anti-debugging, encryption, packing, entry point obscuring, etc.

One of the first methods used to detect malware relied on signatures extracted from the malware body. Despite the significant evolution of malware, using signatures is still the most common detection method used today. However, there are some things that have changed, such as the types of signature data and the methods used to search for such signatures. A modern AV engine may use lots of different types of data extracted from the malware body:

- Patterns, with or without wildcards, also known as 'strings'
- Checksums (CRC, MD5, SHA1)
- Behaviour patterns
- File geometry, execution flow geometry
- Statistic distribution of code instructions

Of course, any combination of the above could be used as a malware signature, and the list is not exhaustive. In an attempt to defeat detection by signatures, malware writers started to use code obfuscation techniques, such as encryption. In the beginning, viruses used fairly simple encryption schemes and only the keys changed from one generation to another, while the encryption algorithm remained constant; these are known as oligomorphic viruses. Later, more sophisticated polymorphic techniques were developed. Such viruses were able to change both the encryption algorithm and the keys used to encrypt themselves upon each replication; some were able to generate multiple encryption layers.

It is still possible to detect a polymorphic virus using signatures, but the virus body must be decrypted first. There are several methods that are widely used in the AV Industry for the purpose of decrypting polymorphic viruses: cryptanalysis (also known as x-ray), dedicated decryption routines, emulation, etc.

2. TECHNIQUES CURRENTLY USED TO DEFEAT POLYMORPHISM

X-ray works by attempting to find the decryption key by using the known decryption algorithm and a fragment of decrypted code, which is part of the signature. For each key, an equation is written, that expresses this key as a function of the encrypted code, decrypted code and the other keys. Solving the system will produce the correct set of keys required to decrypt the virus. The method is fairly simple to implement and has good performance for a single given decryption algorithm. In some cases, it is also able to detect fragments of the virus code, even if the entire virus is not present or is not functional. However, the method does not scale well with the number of detected viruses, because it needs to be run for each different encryption algorithm. Also, it can only handle simple algorithms, as the equation system becomes impossible to solve for more complex ones. Its usefulness is very limited in the case of viruses having multiple encryption layers.

Dedicated decryption routines can be developed to detect any virus, and performance for any given virus is usually better compared to the x-ray method. Unfortunately, writing such a routine requires that the virus is analysed completely and the developer understands completely all the possible variants of encryption that the virus can generate. A thorough analysis of the malware and then developing and testing a specific detection routine could take a lot of work and a lot of time to accomplish. Therefore, the response time when using this solution can often be quite long. Additionally, this method does not scale well with the number of detected viruses, as each file must be checked with all the available routines.

Executing the decryption code from the virus itself would decrypt the virus body with excellent speed performance, but this is not a very good idea, for a number of reasons:

- There isn't any simple, reliable way of stopping the execution when the decryption is complete; therefore the virus could replicate, do some damage to the host system and re-encrypt itself and without being detected
- The virus could do some damage even during decryption
- The code could be buggy and crash or enter an infinite loop
- The virus code could be written for a different hardware/software platform, so it might not run at all

Of course, the code could be executed in a controlled environment, such as a virtual machine, but a general-purpose virtual machine is very complex software. Including one in an AV product would require software emulation for the full system environment, from device drivers to system APIs, which is simply too much overhead.

None of the methods described above is able to detect new malware in a generic way. Their purpose is to just decrypt polymorphic viruses, so that signature-based detection can be used. While it's possible to develop dedicated routines that are able to detect entire malware families (and often new variants as well), writing a routine that is able to analyse an arbitrary program and determine malicious behaviour is not feasible.

Use of emulation solves all the above problems. Potential malicious code runs in a controlled, simulated environment. Hardware resources such as CPU registers are modelled using software data structures. The behaviour of each instruction is reproduced by a set of software routines designed to update the corresponding data structures, in the same way the instructions would update the hardware resources when executed on a real CPU. Each instruction is first decoded, in order to find the instruction type, length, operands that need to be updated, etc. After this, the appropriate emulation routine is called to update the data structure describing hardware resources. The address of the next instruction is obtained either as a result of instruction decoding or computed by the emulation routine (in the case of branch instructions).

The emulation process usually starts at the program's entry point and instructions are emulated sequentially until a malware signature is found, the emulator is able to conclude that the program is not malicious or emulator resources are exhausted. In addition, the emulator has to decide when to scan for malware signatures and what data to scan, call the scanner, collect and analyse data obtained during emulation for behaviour-based heuristic detection or for the purpose of deciding that the program is not malicious.

Emulation can be used to decrypt any encrypted code, regardless of the complexity of the encryption algorithm, given that the decryption code is available and the emulator is provided with enough resources to complete the decryption. Providing resources such as memory is not very difficult, as the requirement is comparable with that of the analysed program (the overhead caused by the emulator's internal data structures is typically negligible). However, emulating code is significantly slower than running the code on a CPU that can execute it natively. This limitation is impossible to overcome, because for each emulated instruction an emulator has to execute hundreds of instructions to decode it, update the internal data structures, decide if scanning is needed, decide if more instructions should be emulated or not, find the address of the next instruction, etc. Typically, emulation is hundreds of times slower than execution.

An emulator in an AV engine is required to analyse any given file in a finite time, during which it must determine if the file is malicious or not. When the maximum allowed time for a file has elapsed and no malware signatures have been detected, the emulator must stop the analysis and conclude the given file is not malicious. It is always possible that the maximum time limit was set to be too short for a particular malware to be detected. On the other hand, increasing this time limit will deteriorate the emulator's average speed. This happens because there will always be a small percentage of clean files for which the emulator will never be able to determine that they are clean, no matter how much it will analyse. Of course, the time limit can be adjusted dynamically: the emulator could increase it if suspicious behaviour is detected or decrease it otherwise. However, there are instances in which legitimate programs are encrypted with the same encryption engines as certain viruses, or viruses use code that looks benign, etc. Even with adjusting the time limit dynamically, there will have to be a hard limit, to avoid having the emulator analysing a file in an infinite loop. This means that it will always be possible for a virus writer to determine what this limit is for a particular AV engine and write a virus that would need to be emulated longer than that in order to be detected. Viruses such as Win32://Coke, KME-based, etc. would take unreasonably long to emulate, but they would still decrypt and replicate on the real machine in a reasonable time, because native execution is typically hundreds of times faster than emulation.

3. DYNAMIC TRANSLATION

The 'dynamic translation' method described below offers the same flexibility as emulation, while improving performance significantly. It relies on disassembling the code to be analysed dynamically and translating it into functionally equivalent code that is safe to execute on the host machine. The executable code obtained as a result of the translation is persisted; if the code is executed inside a loop, the persisted code can, in most cases, be executed directly, without requiring retranslation.

Disassembling and translating an instruction requires a computational effort that is comparable to emulating an instruction; executing the obtained code is typically slower than executing the original instruction, but much less so than emulating the instruction. If a code sequence is executed in a loop, the code will be translated and executed at the first loop iteration, and for all subsequent iterations the persisted code obtained at the first iteration will be executed. Thus, the method eliminates redundant analysis of repeating code sequences. Compared to emulation, the time required to complete the first loop iteration would be approximately the same, while the subsequent iterations will take considerably less time.

3.1. Partitioning the code into blocks

One of the problems that the implementation of the DT engine has to address is determining whether translated code is available for any given instruction, and if so, locating the corresponding code. One possible solution is maintaining a table with virtual addresses of translated instructions and addresses of corresponding executable code. However, searching a virtual address in this table for each processed instruction is computationally very expensive, negating the speed advantage of executing translated code. A much more efficient way of solving this issue would be to partition the original code into blocks of instructions and only store a table entry for each block. This way, the table would have significantly fewer entries and searching would need to be performed for each block, as opposed to each instruction. Dividing the original code into blocks cannot be done in an arbitrary way; blocks need to have some specific properties, as described below, that limit the size of each block. On the

other hand, the bigger the blocks are, the more efficient the storage and searching will be.

For the rest of this paper, a 'basic block' (BB) will be defined as a contiguous block of code having a single entry point at the beginning of the block and a single exit point at the end of the block. If the code within such a block is executed via a call instruction to the beginning address of the block, all the instructions in the block will be executed. A single instruction is needed, at the end of the block, to return control to the caller. As a consequence, any basic block of original code will contain at most one jump instruction. Basic blocks that don't contain any jump instructions are valid, but suboptimal. They could be created as a result of splitting different blocks or for other practical reasons such as the DT engine having insufficient resources to translate a bigger block, etc. Such blocks don't need to be treated differently, as we can consider that they end with a virtual unconditional jump to the following instruction.

Each discovered BB will be described by a set of properties, such as:

- Block boundaries; the address of the first instruction in the BB and the address immediately following the last instruction in the BB will be used to delimit the block; these are linear addresses in the address space of the original code.
- Executable code obtained as a result of translating the original code in the block.
- Miscellaneous flags, indicating if the block was translated or not, if it was scanned for malware signatures, if the block has any known successor blocks, etc.

Discovering and delimiting basic blocks is a dynamic process, meaning that new blocks may be discovered or existing blocks could be modified as a result of processing previously discovered blocks. After translating a block and executing the resulted code, the beginning address of the next block to be processed will be the destination address of the jump instruction at the end of the block that was just executed. Let's call this address the 'current address'. The next block to be analysed has to be a BB starting at the current address. This block will be determined by searching the current address in the list of block address ranges for previously discovered blocks; the following situations are possible:

i An existing BB is found that begins at the current address; this becomes the current BB and will be processed in the same way as the previous BB.

ii The current address is found inside the address range of an existing BB. According to our definition, a BB must have a single entry point; therefore this block must be split. The existing block will be modified such as its address range will end at the current address and a new BB will be created, starting at the current address. If the split block was already translated, the existing code will not be truncated, but the code starting at the current address will be retranslated for the newly created block.

iii The current address is not found inside the address range of any existing BB. In this case, a new BB will be created, starting at the current address.

If the previous BB ended with an immediate jump instruction – for which the destination address is constant – this means that the current BB will always be a successor of the previous BB. This information can be stored, in order to avoid unnecessary searching in the BB address ranges in case these blocks are inside a loop. If the previous jump instruction was unconditional, the current BB is the only possible successor; in the case of a conditional jump instruction, there can be at most two different successor blocks. If a block ends with a computed jump instruction, the destination address could be different each time the block is executed, so in this case determining the successor block requires searching, as described above, in the list of existing block address ranges. A list of successor blocks determined at previous iterations could be also stored and used to speed up searching, in case the number of different possible successor blocks is reasonably small.

Delimiting the original code into basic blocks as described above and maintaining a data structure describing the blocks and the relations between them has several advantages:

- If several blocks are executed inside a loop, searching for a successor block needs to be done only once for each successor; after all the successors of a particular block have been determined, there is no need to search for a successor of that block at any subsequent loop iteration.
- The list of beginning addresses for the discovered blocks can be used as a list of 'entry points' to scan for malware signatures; each discovered BB needs to be scanned only once.
- It provides data for applying dynamic code optimizations. Since optimizing code is computationally expensive, blocks that are executed more frequently are better candidates for optimizations.

3.2. Scanning for malware signatures

After the virus body has been decrypted, the virus can be detected by finding a signature in the decrypted body. If the virus is not metamorphic, any fragment of the virus body could be used as a signature, providing that it is specific enough to provide accurate identification and not cause false positives. Finding a signature in the decrypted virus body raises the problem of deciding when and where to search for signatures; the search method must guarantee that no signature could go undetected, but it also has to be as fast as possible. In the case of a polymorphic virus, the scanner can guarantee detection only if at least one signature search is performed after the virus body has been completely decrypted, but before it starts encrypting itself again.

Given an arbitrary virus, it is extremely difficult to determine the exact moment when the above condition is met during the analysis. This is true for both emulation and the dynamic translation methods. One possible approach would be to detect decryptor loops and scan the code decrypted by each such loop at the end of the loop. However, this is not very easy to implement, as determining decryptor loops, loop exit criteria and range of decrypted code are fairly complex tasks. Also, this method does not guarantee a minimum number of scans; if the virus has multiple decryption layers, or the same layer is executed multiple times (i.e. in brute-force decryption), the code will be scanned redundantly for each decryption layer. There are also polymorphic viruses for which the entire decrypted body cannot be found in memory at any time (i.e. Dark_Paranoid). In such cases, an instruction

or a small code sequence is decrypted, executed and then immediately encrypted again. For these viruses, the decrypted body can be obtained by logging each instruction the first time it is encountered and sorting the logged instructions by address. Scanning the resultant log guarantees that the entire virus code will be scanned for signatures, and no redundant search operations are performed, because no code sequence will be scanned more than once.

It is generally preferable to extract signatures from the constant part of a polymorphic virus and not from the decryptor code; therefore the code that is used for decryption doesn't necessarily need to be scanned. In practice, however, it is typically more computationally expensive to identify the code used for decryption precisely and exclude it from scanning, than it is to scan it.

Another difficulty comes from the fact that one cannot predict the location in the decrypted code where a signature might be present. Therefore, scanning a chunk of code without any knowledge about the code structure requires searching for signatures starting from up to M - N offsets in the given code chunk, where N is the size of the chunk in bytes and M is the size of the shortest signature searched.

Dividing the code into basic blocks, as described in section 3.1., provides a means to scan the entire code that was analysed, in a reliable way and with minimum number of signature searches. Having the data structure describing the basic blocks, it is easy to determine, for any particular block, whether it is the first time we're analysing it (in which case we also need to scan it) or whether we have analysed it before and no scanning is needed. As part of maintaining the BB data structure, we are also required to detect when any block that has previously been analysed is being overwritten (because if this happened, the block would need to be retranslated). A block can be scanned at any time after it was discovered and before it is overwritten, and it needs to be scanned only once. If we choose, for instance, to use only signatures starting at the BB boundary, the maximum number of scans needed will be the number of unique basic blocks in the code. Of course, a signature could span multiple blocks, in which case all of these blocks need to be discovered and decrypted, so that the signature can be detected. If some of the blocks are overwritten before a signature is detected, the code must be logged as described above.

3.3. Translating code

Given an arbitrary program code to be analysed by an AV engine not only needs the code to be considered unsafe, but it could also be the case that the code is compiled to run on a different hardware platform than the AV engine. Even if the code to be analysed executes natively on the same CPU as the AV engine, it might need to run in a different CPU mode (i.e. x86 real mode vs. protected mode), have different memory mapping mode, execute under a different operating system, etc. Therefore, even if the code was safe or we could somehow make sure the host machine cannot be damaged, it still wouldn't be possible to execute correctly any arbitrary part of the given code, from the AV engine process. It is, however, possible to translate the given code into another code sequence that is functionally equivalent with the original one and that can be safely and correctly executed on the host machine. In our case, the host CPU will be used to execute the translated code directly, while in the case of emulation a virtual CPU is used instead of the real one. Other hardware resources (I/O ports, IRQ controllers, disk drives, etc.) are typically virtualized to protect the host machine from any damage.

There are multiple ways in which a code translation that meets the above criteria can be achieved:

i Translating directly from the original code to target code: each original instruction will be decoded and then an equivalent instruction or instruction sequence will be generated for the target code. This is the simplest translation method to implement, for any source and target binary languages.

ii Translating using an intermediate language (IL): each original instruction will first be translated into an intermediate code sequence and then the intermediate code will be translated to target native code. This method is preferable when we have multiple sources (S) and multiple target (T) languages. With direct translation, we would need in this case S * T translators, while using an intermediate language we only need S + T translators (S translators from a source language to IL and T translators from IL to a target language). There are other advantages as well, such as the possibility to perform code optimizations using the IL form. The IL should be platform-independent, but could be designed in a way that favours translation speed for some particular translators (those that are more frequently used). As a drawback, the IL would need to support all the possible operators, operand types and combinations of these, from all the source languages, raising its degree of complexity. This could prove to be a problem, because translating to or from languages with a reduced instruction set is generally faster, per translated instruction, than in the case of complex languages.

iii Combining the above methods could be achieved in a way that preserves the advantages of both, without any of their disadvantages. Most instructions could be translated using a fairly simple intermediate language, while the most exotic and complex ones, that would also require a complex IL, will be translated directly.

Typically, the code obtained as a result of translation will not be as efficient as the original code. This may happen for various reasons: some instructions or operand encodings from the source language might not have a 1:1 correspondence in either the IL or the target language, some hardware resources used in the source language need to be preserved or not used at all in the target language because of specific restrictions, memory mapping has to be virtualized because the source and target languages might use different mappings, etc.

It is possible to perform some optimizations at basic block level, at translation time, to improve the efficiency of the translated code. If the translation uses an IL, the best idea would be to perform the optimizations on the IL, because the algorithms involved will only need to support this language, as opposed to all the source and target languages. The IL could also be designed to facilitate optimizations such as define-usage chains, copy propagation, etc., while the other languages might not be very suitable for performing such optimizations. Also, the IL may be used to pass translation 'hints' from the source translator to the target translator.

If the analysed code is linear, the code will be translated once and executed once. As the execution time is negligible compared to translation time, in this case translation would account for most of the analysis time. For clean files that don't unpack or decrypt themselves and don't have any suspicious behaviour, there is usually no need to analyse lots of looping code; in this case, the translators must be optimized for speed. On the other hand, for polymorphic malware and occasional clean files that contain unpacking or decryption code, most time will be spent analysing loops and executing code that is already translated. In this case, the speed of the code generated by the translators is more important than the translation speed. As improving the efficiency of the translated code is done at the cost of translation speed, a compromise between these two must be obtained.

An example of code translation is given in Appendix A.

3.4. DT execution flow

For each file that needs to be scanned for malware, analysis consists of sequentially identifying and processing of basic blocks, as defined in section 3.1. At the beginning of the analysis, the current address is initialized to the entry point of the program to be scanned. After each BB is processed, the current address is updated to the destination address of the jump instruction at the end of this block. The analysis continues with the BB starting at the new current address, until a malware signature is detected or the program is determined to be clean.

In a simplified description, after each BB is analysed, the following processing algorithm has to be performed for the next one:

1 Look in the data structure describing relationships between blocks for a known successor of the last analysed block; if a known successor exists, make this block the current BB and continue from step 6.

2 Search the block address range table for a previously discovered block starting at the current address. If such a block is found, make it the current BB and continue from step 5.

3 If the current address is found inside the address range of an existing BB, split this BB to end at the current address. Create a new BB starting at the current address, make the new block the current BB and continue from step 5.

4 If no existing block was found, that either starts or includes the current address in its address range, create a new BB starting at the current address; this will be the current BB.

5 Update the data structure describing relationships between blocks: store the information that the current BB is a successor of the previously processed BB.

6 If the current BB was not scanned for signatures, scan for signatures starting at the current address. If a signature is found, stop the analysis and report the file as malicious, otherwise mark the block as scanned.

7 If the current BB is already translated and the code was not overwritten since last translation, continue from step 10.

8 If the current BB was previously translated but the original code was overwritten, discard the translated code, as it is no longer valid.

9 Translate the current BB.

10 Prepare for executing the translated code for the current BB: save hardware resources that are used by both this algorithm and the translated code and need to be preserved, such as CPU registers, etc., depending on implementation.

11 Execute the translated code for the current BB via a call instruction; after execution is complete, control will be returned by the executed code to the caller.

12 Restore resources saved at step 10.

13 Handle any errors that might have happened during execution, decide whether to continue analysing the next block or stop.

During execution, the translated code has to check, before each write to memory operation, whether an existing block would be overwritten by the data being written. The checking may be skipped only if it can be determined, at translation time, that this particular write operation could never overwrite an existing block. If one or more blocks are overwritten, the corresponding translated code will be marked as 'dirty', meaning that it will be discarded at step 8, when those blocks will be processed again. If one of the blocks that were overwritten is the current BB and code beyond the current execution point was overwritten, the execution must not be allowed to continue to the end of the current BB, as this would mean executing code that is no longer valid. In this case, the write operation is allowed to happen and then execution is allowed to continue until the entire translated code sequence corresponding to the current original instruction is executed. This is done because execution cannot be interrupted in the middle of the code sequence for an original instruction, otherwise resuming the execution would not be possible. After all, overwritten blocks are marked as dirty, execution is interrupted and control is returned to the caller, with the current address set to the address where execution was interrupted.

Handling of exceptions such as division by 0, page fault, etc. is done in a similar way: execution is interrupted at the address of the instruction that generated the exception; if an exception handler is present, execution may continue with the exception handler code. Information is passed to the exception handler, enabling it to resume execution at the point where it was interrupted.

Obtaining the address of the original instruction corresponding to a given instruction in the translating code requires some computational effort. For the original code, the real CPU keeps an instruction pointer register, updating it after each instruction is executed. Reproducing this behaviour in the translated code would mean generating and executing code for an extra

```
<add instruction_pointer, instruction_code_size>
```

operation, for each translated instruction. For simple arithmetic instructions, this would mean doubling the translated code size and execution time, which is unacceptable. Therefore, computing the value of the current instruction pointer will only be done when needed, if an exception happens or the current block is being overwritten.

For this purpose, a table is used, that relates offsets of instructions within the current BB to addresses of corresponding instructions, in the original code.

Special care has to be taken when calling high-level language compiled functions from the translated code. CPU registers that might be used by the compiler in global optimizations must be saved at step 10 and restored before calling a high-level language function, in case they were changed by the translated code. The stack pointer and frames must also be preserved. The translated code must also save and restore any registers computed before the high-level language function is called, whose values are still needed after the function returns.

3.5. Environment

In order to obtain a correct behaviour while analysing programs, the DT engine must provide access to various hardware devices (disk drives, keyboard, mouse, network interface, video card, real-time clock, etc.), as well as software resources, such as BIOS data structures and routines and operating system APIs. If the code provided as input to the DT engine could be malicious, most of these resources need to be virtualized. This requires a lot of development effort, given the large number of devices and system APIs that need to be supported. However, virtualizing the system environment for a dynamic translation engine is done in almost the same way as in the case of an emulator, so code reusing is possible if an emulator is available.

Accessing virtualized devices, as opposed to real ones, offers improved speed performance. A virtual device is in fact a data structure in memory, which can be accessed much faster than a disk drive, for instance. Often there's no need to fully implement all the functionality of a device, because no existing malware would require it. Therefore, a virtual device will be less complex than a real one, making it even faster in operation.

Real devices may be used in a few cases – for instance, the current time could be obtained by using the real-time clock. However, this may cause the analysed code to behave incorrectly. As the code actually runs slower inside the DT engine than it would run natively, time inside the DT engine should also pass proportionally slower. Otherwise, it would be possible for the analysed code to determine, based on this inconsistency, that it's running in a virtual environment. This is used by malware as an anti-emulation technique.

Some malware would function correctly only if the environment is configured in a particular way. For instance, they might require a specific OS version, work only on a certain file system type, check for the presence of a specific file in a known place or work only within a certain calendar date range, etc. In these cases, it is difficult to configure the environment so that the code will run correctly, as different malware may have different conflicting requirements.

4. DEVELOPING FURTHER APPLICATIONS

Decrypting code in order to provide signature-based detection for polymorphic executable file infectors is just one possible application of dynamic translation. The code to be analysed doesn't need to be a CPU instruction code, it could also be a platform-independent byte code, such as MSIL byte code or a script. Translating from a non-binary language to native executable code for the host CPU can be done in a similar fashion as translating from a binary language. Using an intermediate language might be more suitable for translating from a non-binary language, because it could provide support for variable number of operands and operand allocation, which are required by virtually all script languages.

Using DT to analyse scripts, however, has to deal with some specifics. For instance, a script might not be able to modify itself while it is being interpreted, but it could generate other script files dynamically and call the interpreter to run these. In this case, each particular file can be translated statically, but the analysis is still a dynamic process, because not all of the component files are available at the beginning of the analysis. Providing an environment for analysing scripts is more difficult than in the case of executable files. Some of the major challenges are accurately reproducing the behaviour of the script interpreter and of OS commands and utilities that might be called from the script. Different versions of operating system or script interpreter might behave differently.

Dynamic translation can also be used to translate malware detection routines from a platform-independent byte code into executable code for the host CPU. This way, an AV engine can easily be updated with new detection capabilities, without the need to actually change the engine code. The benefits provided are smaller engine code, smaller (and faster) updates and less testing effort required for the new routines. The new detection routines will be sent as 'data', the same way as signature/pattern database updates. They will be loaded and translated to native code when the engine is loaded in memory and then executed, as needed, with little speed penalty compared to native code generated by a compiler.

The approach used to translate such routines will be slightly different from the one used to translate files that are scanned for malware, because in this case it is already known that the detection routine code is not malicious. This allows the use of a real environment as opposed to an emulated one. Also, there is no need to scan any of the translated code and there is no need to check for overwritten blocks, because the code doesn't decrypt or otherwise modify itself. All the successors for all basic blocks can be determined when the engine is loaded, which means that we don't have to search for any successor block at execution time. For these reasons, the translated code obtained for detection routines will be a lot more efficient than the code typically obtained by translating files for the purpose of scanning.

In the case of packed executable malware, unpacking is typically needed before a signature can be detected. Using signatures extracted from packed code is not always practical, even for non-polymorphic malware. Some strings or other data might change inside the binary, upon each replication, causing the packed file to change in such a way that the signature won't match any more. Detection based on packed code is not possible if the packer is polymorphic. Heuristic or generic detection can be achieved only by using unpacked code.

Writing unpacking routines for all the packers publicly available takes a lot of development and test effort. In some cases, writing an unpacker routine would require reverse engineering the packer and there might be some legal restrictions preventing this. In the absence of a dedicated unpacking routine, a packed executable could be emulated until the unpacked code is obtained. However, unpacking with

an emulator could be very slow, especially for large packed files that would typically require emulating several millions of instructions. Using dynamic translation, a file could be unpacked significantly faster, compared to emulation, providing detection for malware packed with new packers, with reasonable speed performance, before a dedicated routine is developed. In some cases, the generic unpacking using DT could prove fast enough that dedicated routines won't even be needed.

APPENDIX A

The table below shows an example of translation of a 16-bit x86 code sequence sample to 32-bit x86 target code, as generated by the prototype implementation of the Dynamic Translation method.

Original code	Intermediate language	Translated code	Comments

L_decrypt:
mov eax, [bx]
	mov -, d[(reg.DS<<4)+reg.BX], reg.EAX	movzx edx, w[ebp+offset_reg_BX]
		movzx eax, w[ebp+offset_reg_DS]
		shl eax, 4
		add edx, eax
		call [ebp+offset_memory_mapper]
		mov ecx, [eax]
		mov [ebp+offset_reg_EAX], ecx
			The memory mapper receives the virtual address in edx and returns the real address in eax

add ax, 1234h	add reg.AX, 1234h, reg.AX	lea eax, [ebp+offset_reg_AX] add w[eax], 1234h	Possible optimization: add w[ebp+offset_reg_AX], 1234h

xor al, ah	xor reg.AL, reg.AH, reg.AL
	Saveflags reg.Flags
		mov eax, [ebp+offset_reg_AH]
		xor [ebp+offset_reg_AL], al
		lahf
		seto al
		mov [ebp+offset_reg_Flags], ax
			The layout of the Flags register image in memory may be different than the actual layout of the native Flags register, for speed reasons

mov dx, ax	mov -, reg.AX, reg.DX	lea eax, [ebp+offset_reg_DX]
		movzx ecx, w[ebp+offset_reg_AX]
		mov w[eax], cx
			Possible optimization: movzx ecx, w[ebp+offset_reg_AX] mov w[ebp+offset_reg_DX], cx

shr eax, 16	shr reg.EAX, 10h, reg.EAX	lea eax, [ebp+offset_reg_EAX] shr d[eax], 10h	Possible optimization: shr d[ebp+offset_reg_EAX],10h Flags don't need to be stored, because they will be overwritten by the next instruction.

ror ax, cl	ror reg.AX, reg.CL, reg.AX
	Saveflags reg.Flags
		mov cl, [ebp+offset_reg_CL]
		ror w[ebp+offset_reg_AX], cl
		lahf
		seto al
		mov [ebp+offset_reg_Flags], ax
		movzx edx, w[ebp+offset_reg_BX]
		movzx eax, w[ebp+offset_reg_DS]	

mov [bx], ax	mov -, reg.AX,
	w[(reg.DS<<4)+reg.BX]
		shl eax, 4
		add edx, eax
		call [ebp+offset_memory_mapper]
		mov cx, [ebp+offset_reg_AX]
		mov [eax], cx	

mov [bx+2],dx	mov -, reg.DX,
	w[(reg.DS<<4)+reg.BX+2]
		mov edx, 2
		add dx, [ebp+offset_reg_BX]
		movzx eax, w[ebp+offset_reg_DS]
		shl eax, 4
		add edx, eax
		call [ebp+offset_memory_mapper]
		mov cx, [ebp+offset_reg_DX]
		mov [eax], cx
			The virtual address could be obtained by adding 2 to the virtual address computed for the previous instruction. However, implementing such optimizations in a generic way is computationally expensive.

Original code	Intermediate language	Translated code	Comments
inc bx	Loadflags reg.Flags inc reg.BX, -, reg.BX Saveflags reg.Flags	mov ax, [ebp+ offset_reg_Flags] add al, 7Fh sahf lea eax, [ebp+offset_reg_BX] inc w[eax] lahf seto al mov [ebp+ offset_reg_Flags], ax	The previous 'mov' instructions in the original code do not affect CF, but the corresponding translated code sequences do so. The IL only allows saving all flags and not individual ones, so we must load the native flags prior to executing the 'inc' instruction that doesn't affect CF, in order to preserve the correct value of the CF flag.

loop L_decrypt
L_endloop:
	sub addr_L_endloop, (reg.CS<<4), reg.IP
	
	dec reg.CX, -, reg.CX
	setz -, -, reg.T32
	
	jopz reg.T32, (reg.CS<<4)+reg.IP–24h
		lea eax, [ebp+offset_reg_IP]
		mov ebx, addr_L_decrypt
		movzx ecx, w[ebp+offset_reg_CS]
		shl ecx, 4
		sub ebx, ecx
		mov [eax], bx
		lea eax, [ebp+offset_reg_CX]
		dec w[eax]
		lea eax, [ebp+offset_reg_T32]
		mov ebx, eax
		mov eax, 0
		setz al
		mov [ebx], eax
		mov ecx, [ebp+offset_reg_T32]
		jecxz L_jump_taken
		mov b[ebp+offset_jump_info], 2
		jmp L_return
		L_jump_taken:
		mov edx, -24h
		add dx, [ebp+offset_reg_IP]
		movzx eax, w[ebp+offset_reg_CS]
		shl eax, 4
		add edx, eax
		mov [ebp+offset_crt_address], edx
		mov b[ebp+offset_jump_info], 3
		L_return: ret
			The code sequence for a jump instruction must update the instruction pointer register, compute the virtual address of the next instruction and provide the DT engine information about the jump instruction, such as type of jump and whether the jump was taken or not. It is possible to replace 'jmp L_return' with a 'ret' instruction, but this would negate the possibility of calling a debug trace function from the generated code after each instruction sequence, making debugging of generated code more difficult. The final 'ret' instruction is not generated while translating the jump instruction; it is generated after the entire block has been translated. This ensures that any translated block will be able to return control to the caller, even if the original code in that block doesn't end with a jump instruction.

The following format was used for IL instructions: <opcode source_operand_1, source_operand_2, destination_operand>. For example, 'add x, y, d' means 'd = x + y'. Loadflags and Saveflags are not separate IL instructions; they are encoded in the binary form of affected IL instructions. In this particular implementation, the IL language accepts simple operands, such as registers and constants, as well as more complex operands, like registers shifted with a constant, a sum of register, constant or shifted operands, etc. It is possible to design the IL to only support simple operands, in which case each IL instruction would be simpler and faster to generate, but more IL instructions would be needed, in average, to translate an original instruction.

APPENDIX B

The following table shows comparative speed test results, obtained by benchmarking a prototype implementation of the dynamic translation method presented in this paper, versus the emulator in the last version of *RAV* anti-virus engine. Within each test, the DT prototype and the emulator analysed the exact same instructions and scanned for malware with the same signature set. The best time of three consecutive runs was selected, for each test.

The files used for the first three tests were infected with polymorphic file infectors; both the emulator and the DT prototype had 100% detection rate on these test sets. The test set for the last test consisted of 100 copies of an executable file containing a nested decryptor loop – interior loop having 1,020 iterations, exterior loop having 256 iterations.

The benchmark results indicate that:

- The DT method provides a significant speed improvement over emulation, in all tests.
- In the case of emulation, speed performance is not affected, in most cases, by the complexity of the analysed files. The time required to emulate a given code sequence is proportional with the type and number of instructions emulated. An emulator takes little or even no advantage of repeating code sequences (a slight improvement is noticed for the last test set).
- The speed performance provided by the DT method, relative to native execution, improves with the average number of analysed instructions per file, as the probability of finding repeating code sequences increases. Thus, detection of heavily polymorphic viruses, requiring millions of instructions to be analysed, can be accomplished in a reasonable time by using dynamic translation.

test number	method	analyzed instructions x 1,000,000	analyzed files	average instructions per file x 1000	total time [seconds]	MIPS P4@3GHz	average slowdown rate*
1	emulation DT	394	7070	58	63 15	6.3 26	317 77
2	emulation DT	422	2000	211	65 11	6.5 38.3	307 52
3	emulation DT	427	738	579	68 8	6.3 53.4	317 37
4	emulation DT	418	100	4180	54 7	7.7 60	260 33

* the average slowdown rate is defined as the time needed by emulation / DT to analyse a given test code divided by the time needed by a real CPU to natively execute the same code. An average of 1.5 clock cycles per instruction was used for the purpose of estimating the total time required for native execution of the files in the test sets. Actual execution time would be difficult to measure given that infected files were used.

