The Design Space of Metamorphic Malware

Andrew Walenstein[†], Rachit Mathur[‡], Mohamed R. Chouchane[†], and Arun Lakhotia[†]
[†]University of Louisiana at Lafayette, Lafayette, LA, U.S.A.
[‡]McAfee Avert Labs, Beaverton, OR, U.S.A.
arun@louisiana.edu
rachit_mathur@avertlabs.com
mohamed@louisiana.edu
walenste@ieee.org

Abstract: A design space is presented for metamorphic malware. Metamorphic malware is the class of malicious self-replicating programs that are able to transform their own code when replicating. The raison d'etre for metamorphism is to evade recognition by malware scanners; the transformations are meant to defeat analysis and decrease the number of constant patterns that may be used for recognition. Unlike prior treatments, the design space is organized according to the malware author's goals, options, and implications of design choice. The advantage of this design space structure is that it highlights forces acting on the malware author, which should help predict future developments in metamorphic engines and thus enable a proactive defence response from the community. In addition, the analysis provides effective nomenclature for classifying and comparing malware and scanners.

Keywords: Metamorphic Malware, Virus Scanner.

1. Introduction

Metamorphism is the ability of malware to transform its code. This ability was first introduced in viruses and was later used by worms, Trojans, and other malware. There now exist several metamorphic engines—programs that implement only the logic for transforming code—that can simply be linked to any program to make it metamorphic. Metamorphic malware can be classified into four broad groups defined by two parameters. First, metamorphic malware may be either closed-world or open-world. Closed-world metamorphic malware is malware that is self-contained; in contrast, open-world malware may extend its capability by communicating with the world, say by downloading plug-ins from the web. To our knowledge, all metamorphic malware thus far has been closed-world, though open-world engines have been hypothesized (Driller 2002; Z0mbie; Ször and Ferrie 2001). Second, metamorphic malware may be either a binary-transformer or an alternate-representation-transformer. The former class transforms the binary image that is executed, whereas the latter class carries its code in a higher level representation, which is used for transformation.

Metamorphic engines, historically, have been written primarily by self-trained individuals, and have tended to be complex and buggy. However, with nations adopting cyber-warfare doctrines, one can anticipate an organized effort to develop state-of-the-art robust metamorphic engines. Besides, even teenaged criminals have begun to make a handsome, albeit illegal, living from their craft and hence have an incentive to improve the potency of their wares. Thus, it would be reasonable to predict that, in the not too distant future, one would see highly stable metamorphic malware in the wild. This paper takes a proactive stance in dealing with closed world binary transformer metamorphic malware by analyzing its design space. It restricts attention primarily to the design choices relevant to scanners that use only statically extractable information for detection.

Design spaces are frequently formalized as a collection of goals, actions and constraints, and their implications or tradeoffs. Our design space analysis is from the view of the malware author, whose primary goals are to have their malware function as intended, to effectively replicate, and to avoid recognition by scanners. This analysis differs from prior treatments in that, instead of enumerating design solutions in a relatively decontextualized way, it situates design choices in the context of the malware designer's goals. It also traces tradeoffs between these choices and how their interactions with one another affects the design goals of metamorphic malware. Making the design space explicit in this way offers several benefits, including: providing ways of fundamentally changing discourse by allowing discussion of abstract classes of malware designs, and laying the

groundwork for predictive models for future metamorphic engines by situating author options in the context of their goals.

As is with any warfare technology, metamorphism has defensive applications as well. Dube, Edge at al. (2006) have investigated the use of metamorphism as a means of protecting software. With software rapidly becoming one of the key elements of modern defense, including warheads and bombs, methods to protect software takes on a different level of significance. Indeed, Edge, Dube et al. (2006) explicitly studies the protection mechanisms used by computer viruses and how they may be used for protecting software. While the discussion in this paper considers the design choices for an adversary using metamorphism, it is obvious that the same set of choices is equally applicable when metamorphism is used for defense.

The paper is structured as follows. Section 2 introduces the prototypical anatomy of closed-world, binary-transforming metamorphic malware. This anatomy is used to structure our model of the design space for such metamorphic malware. This space is outlined in Section 3. Section 4 lays out the design space by traversing through the anatomy and listing options designers have for implementing their goals, drawing implications and tradeoffs arising from these options along the way. Section 5 concludes the paper by summarizing the advantages of the structure, and by listing directions for future work.

2. Prototypical anatomy of metamorphic malware

The classes of design options available to a malware author are related to the logical structure of the metamorphic programs they create. Thus, in order to organize a presentation of design choices, this logical structure must first be described. Our model of the anatomy of a prototypical closed-world, binary-transforming metamorphic engine is constructed as an analogue to Singh's anatomy of a computer virus (Singh 2002). It is more granular than that introduced by Lakhotia *et al*. in (Lakhotia, Kapoor et al. 2005), however. The anatomy of a metamorphic engine identifies functional units (or organs) which are conceptual units of tasks that must be performed by the engine but are not required to be implemented as modules or functions. Figure 1, adapted from the model of Lakhotia et al. (Lakhotia, Kapoor et al. 2004), depicts our simplified model of the anatomy. The features of the model are as follows.

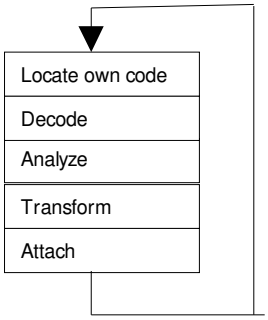


Figure 1: Anatomy of a metamorphic engine

Locate own code. A metamorphic engine must be able to locate the code to be transformed each time it is called upon to transform it. Parasitic metamorphic malware which transforms both its own code and its host's must be able to locate its own code in the new variant.

Decode. The metamorphic engine will need to decode information that is required to perform the transformations. A key piece of information is the metamorphic program itself since, in order to transform itself, some representation of itself is needed so that it knows how to make the transformations. In essence, this involves disassembly. However a metamorphic engine may also need to decode other types of information it requires for analysis or transformation. The information can be encoded in the malware body, either in the data segments, or in the code itself. Choice including using flags, bit-vectors, markers, hints, and so on. For example, the Miss Lexotan engine encodes register liveness information by storing it in the second argument of an "`xor ebp,`

`immed`" instruction (Vecna 1998). Ordinary unpacking or decrypting of the whole executable is not within the scope of this Decode organ.

Analyze. In order for the metamorphic transformations to work correctly, certain information will normally need to be available. For example, certain transformations are semantics-preserving only when a given register is known to be *dead.* In order to perform these transformations correctly, then, the engine must have register *liveness* information available. When such information is not made explicitly available (and hence decoded), it must be constructed by the engine itself. In the case of register liveness, this can be constructed, in part, via a "def-use" analysis. The control flow graph (CFG) of the program is one piece of information that is frequently required for analysis and transformation. It may be used, for example, to rewrite the control flow logic of a program if a transformation expands the size of code.

Transform. The Transform step transforms the malware code into equivalent code, usually by replacing instruction blocks in the code to be transformed with equivalent ones. Well known metamorphic transformations include register-renaming, code substitution, NOP insertion, garbage insertion, and instruction reordering within a block.

Attach. Parasitic metamorphic malware need to attach the new version to a host file.

The ordering of the organs represents the direction of information flow, which is not required to be the execution order. The feedback loop acknowledges that the output of a metamorphic engine may become its input in the next generation. This feedback has previously been deemed as the Achilles' heel of a metamorphic virus (Lakhotia, Kapoor et al. 2004). It is critical for understanding the implications of design choices and the tradeoffs that typically must be made during engine design.

3. The structure of the design space

There are possibly many ways of constructing useful models of a design space such as the one facing authors of metamorphic engines. Significantly, a design space is more than just a listing of possible solution features: it must relate to resolution of designer's goals, and make explicit the tradeoffs implicit in various solution strategies. We define a model of the design space for metamorphic malware in terms of *goals*, *options*, and *implications*. Goals are things the designer intends to achieve. Options are ways of achieving goals and typically have both negative and positive implications. Implication analysis follows options to how they affect other options or goals. We therefore treat the design space as a graph of design choices connecting goals and other design choices via design implications.

Goals. The typical goals of metamorphic malware are to thwart reliable detection (even when the scanners are given one of its variants), propagate effectively, and perform its intended functions. Another important goal common to malware authors is simplicity or parsimony, i.e., they usually wish to choose the simplest and easiest option for achieving their other goals. There are two main goals relating to thwarting detection. First and foremost, the central purpose of metamorphism is to reduce the number of statically identifiable patterns in that can be used for signature detection. Second, the author may wish to make it difficult for scanners or human analysts to properly analyze its code.

Options. The key goals of a metamorphic malware author are in thwarting detection by reducing the number of identifiable patterns, and by raising the level of difficulty for analysis. For the former purpose, program transformations are selected that alter the form of the code without affecting its functionality. In general, the aim is for maximal change in form between all variants created. For the latter purpose, obfuscating transformations can be used, or transformations can be selected that require the solution of problems too difficult for scanners to perform. In the general case, the obfuscations may make only minimal impact on the form but still disable recognition by scanners. Key design choices available to the metamorphic malware designer relate to how these transformations are implemented. Any metamorphic engine must have access to information about the code to transform. This information allows the engine to determine whether or not some

program transformation can be performed at a given point in the code. For the purposes of this closed-world engine design space, there are three possible ways of furnishing this information: (1) encode it into the malicious program itself, (2) assume it, or (3) construct it through self-analysis. These strategies yield multiple options, depending upon what the information is needed for. These options are enumerated in a principled manner by consulting the anatomy of the metamorphic engine from Section 2.

Implications. The complexity of the design space is greatly affected by how the goals, options, and implications interact. In a simple design space no design choice restricts or interferes with another (i.e., there are no cross-cutting constraints), no tradeoffs need to be made, and no options impact multiple goals. In a complicated design space there are such interactions and tradeoffs. Metamorphic malware presents an especially complicated design space because of the "meta": the space is a recursive one—one in which design choices affect prior design choices. If the metamorphic engine did not need to transform its output, the space would be less complicated. Instead, because it must transform its own output, a single design decision may create many implications, both positive and negative.

Consider an example. A designer's goal may be to defeat disassembly by malware scanners. One option would be to insert jumps over junk bytes, which can defeat so-called “linear-sweep” disassembly. However this decision has consequences: now the malware itself cannot use linear-sweep disassembly in its own analysis. Inserting junk bytes is one option, but notice the design space is not understood until the implications are elicited and placed within the context of the designer's goals.

Some initial remarks about the options are required before continuing. Recall that we listed three different strategies for defeating anti-malware analysis: removing consistent patterns, defeating analysis methods through obfuscation, and requiring scanners to solve difficult problems. Ordinarily, the latter type of attack is possible because the engine is able to more readily access or compute information needed to perform the analysis and transformations. A scanner may not always be able to construct the same information. For example, a metamorphic engine may capitalize on the existence of problems which are widely believed to be hard on average. These problems are currently relied upon in many cryptographic schemes and can be used by a metamorphic engine to introduce obfuscations only it can efficiently solve (using some sort of "certificate" or "extra knowledge") and thus be able to efficiently analyze its code in order to transform it. A metamorphic engine may also capitalize on the existence of problems which are widely believed to be hard in the worst case. These include the NP-complete problems. Coming up with “efficient exact” solutions to these problems is an immediate ticket to fame and fortune and is obviously not a reasonable expectation of a generic anti-malware scanner. Designers of anti-malware scanners can at most apply “efficient approximate” solutions to such problems (or efficient exact solutions to their sub-problems).

4. The design space

Given the anatomy of a metamorphic engine and the formula for its structure the design space can be present. The following sections list some of the significant design choices in the space. It is organized by the organization of organs in the prototypical malware.

4.1 Locate own code

Parasitic metamorphic malware must locate its own code in order to transform it. There are two main possibilities for transformation: keep it in a fixed location (the host executable's entry point), or change its locations. While the former option satisfies goals of simplicity, it does little to advance the goal of thwarting detection. The latter forces the engine to discover the locations. One option is to encode it; frequently this is done by using special markers or code segments known to indicate the beginning of the malware code. The marker can remain constant across generations of the malware or it can be transformed. There are tradeoffs to either choice. If the marker is required to remain constant across generations, then the transform step may not transform it. This is obviously the easiest choice for the engine, but it leaves it open to detection techniques which look for that

specific marker. If the marker is transformed by the engine, simplicity is traded off since the engine now needs some method for creating new marking techniques programmatically.

4.2 Decode

Information may be encoded in the malware body to enable it to successfully carry out any of the tasks identified in its anatomy. To further harden the task of malware scanners, the engine could deliberately require the scanner to "answer hard questions" at any of these steps. We use the word "hard" here very loosely to refer to problems which are unsolvable (e.g., the Halting problem), *known* not to have an efficient exact solution (e.g., provably intractable problems), *believed* not to have an exact efficient solution (e.g. NP-complete problems), or simply whose known solutions still require an unacceptably high amount of space or time to carry out their computations on modern computers. These questions, of course, should not be as hard to answer for the engine.

To circumvent the hardness of some general question, the engine may carry—or otherwise know how to infer from the code—some extra information that enables it to quickly answer this question and move on to the next functional unit and eventually be successful in efficiently generating a semantically equivalent variant of the malware. The Miss Lexotan virus, for example, embeds register liveness information in the code to be transformed so that it can transform the code without the need to perform an imprecise and inefficient def-use analysis on the entire program. But if the embedded information is constant it simplifies the scanner's problem. Transforming the encoding during replication adds complexity to the engine and may restrict the set of transformations it can successfully apply.

The simplest disassembly technique, the one commonly used by disassembly tools and debuggers, is linear sweep (Linn and Debray 2003). While meeting the goal of simplicity, the choice of using linear sweep significantly limits the code for which the malware can yield correct disassembly. For instance, for architectures such as the Intel IA32, linear sweep cannot correctly disassemble programs with code and data interspersed, with data bypassed using jump instructions (Z0mbie; Lakhotia, Kapoor et al. 2004). Another option is to use other, more powerful, disassembly methods such as recursive traversal or knowledge-based disassembly. They both expand the class of programs that can be disassembled, but still may not yield correct results with programs using jump tables for indirect transfer of control (Linn and Debray 2003).

The choice of a disassembler has several important implications. An important *overall* effect is that it can limit the transformations that the metamorphic engine may perform. Since the malware needs to disassemble its own code, it cannot afford to use obfuscating transformations that thwart its chosen disassembly method. The disassembler choice can also constrain the choice of variation-inducing transformations. For example, using a linear sweep method may simply rule out adding variety to the control logic via jump tables. The choice of a disassembler also has an effect on the set of target host files that a metamorphic virus can infect. For instance, a metamorphic engine that does not disassemble floating-point instructions will not be able to correctly disassemble any payload that contains such instructions. As a result such programs cannot acquire metamorphic capabilities by linking to that engine. Metamorphic malware can also deliberately use anti-disassembly obfuscation techniques that only it can quickly circumvent, normally because of assumptions it makes. The opcode shifting metamorphic transformation was shown by Dube et al. (Dube, Edge et al. 2006) to be quite effective at thwarting disassembly of the malicious code.

4.3 Analyze

Metamorphic engines have several options to choose from to construct the information needed for successful transformation and attachment. As described in Section 3, the designer can choose not to provide the information, or can choose to assume properties, encode the required information, or extract it.

Provide no information.

Choosing not to provide the information will restrict the designer's options for transformations. For example, one of the most commonly required pieces of information is a control-flow graph (CFG).

A CFG is required to "rewire" the control flow if the starts of basic blocks are moved during transformation. So if a CFG is not available the malware author may not, in general, be able to add transformations that shrink or grow the program. Similar information, including register liveness and block-boundary information, are also commonly needed during transformation. Without them the engine designer is analogously restricted in the payload and engine code, and in the choice of transformations relating to registers and code blocks. This restriction is felt particularly keenly when implementing *conditional transformations* (see Section 4.4).

Restrictions on the transformations normally constrain the effects of metamorphism, reducing the variation and simplifying the problem of scanning. For example, without liveness, many expression transformations are not feasible, and then compiler techniques such as constant folding might be used to counteract variation-inducing transformations, thus assisting the scanner. In general, A number of other exact and approximate techniques were developed to assist anti-malware scanners in detecting metamorphic malware that uses these techniques. Code normalization techniques have been proposed to counter the code substitution and instruction reordering transformations (Lakhotia and Mohammed 2004; Bruschi, Martignoni et al. 2006; Walenstein, Mathur et al. 2006). Techniques based on analyzing the structure of the malware's code and its control-flow graph have also been studied to counter the effects of these and other transformations (Kruegel, Kirda et al. 2005; Bruschi, Martignoni et al. 2006) (Jha, Seshia et al. 2005).

Assumptions.

The designer can choose to encode certain assumptions within the engine, obviating the need for the information. Since code is written by the malware author herself, she can ascertain that the assumptions hold throughout her code. This choice simplifies construction, but restricts the code to only that which follows the assumptions. Furthermore, the choice may also make certain transformations more difficult, or even impossible. For example, if Analyze and Transform steps make use of the property "register `eax` is not live after a `push eax`", it needs to be preserved for it to work correctly again in its offspring. Thus the Transform organ must generally ensure that this property holds across all variants of the malware.

Encoding.

Hints or data may be encoded using a certain pattern of code. For instance, as it propagates, metamorphic malware can also carry a data structure encoding the malware's full CFG. Alternatively, it might encode only some hints in the code that tells it how to obtain the targets of an indirect jump or call instruction. The metamorphic engine can use the CFG for its analysis, transform the code, and create an encoding of the new CFG. The construction of the initial CFG can be done with special tools, or manually, before the malware is released.

Such an encoding of information can make analysis a breeze for the engine but not for an anti-malware scanner that is not aware of the existence or significance of the hint. For example, as mentioned before the Miss Lexotan engine encoded liveness information in special instructions. For any scanner not aware of these, a time consuming def-use analysis is required on the entire code. Moreover, such analysis is usually imprecise and hence unhelpful in detecting the inserted garbage code. More generally, metamorphic engines which encode information needed in transformation succeed in performing semantics-preserving transformations correctly because such engines know how to interpret (and what to infer from) such encodings. Furthermore, the engines can introduce obfuscations that cannot be deobfuscated by a general deobfuscator. The engine would use such encodings (which are unintelligible to a general scanner) to understand its obfuscations. This constitutes an explicit attempt by the engine to defeat deobfuscation. Current engines do not appear to explicitly try to trip deobfuscators. They carry additional information (or assumptions) so they can do their own work correctly.

Extraction.

Extracting the information frequently means implementing complicated analysis routines; these require skill and are difficult to correctly implement. They run counter to the designer's goals of parsimony. Several implications are created by the choice of information to extract.

Choosing to extract a CFG can place other restrictions on the type of code that can be used in the payload or engine, and can restrict the range of transformations it can make that affect control flow. Ignoring the impact of exceptions or signal handling, the completeness of the CFG depends on the correct decoding of jump and call instructions. There are two types of jump and call instructions: direct and indirect. In the direct category are instructions in which the address to which control is transferred is available in the instruction itself. Indirect jump/call instructions are those which obtain

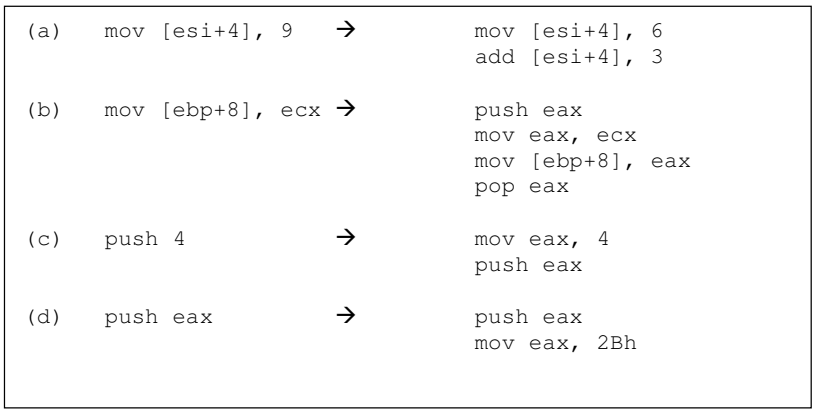


Figure 2: Conditional and Unconditional Transformations

the address for transferring control from either register or memory location. CFG construction is easy if only direct transfers of control are used. When indirect jumps are present, it may be extremely difficult to discover all possible branch targets; in the worst case, it is not solvable. Thus the decision often involves a tradeoff involving three issues: simplicity or ease in analysis, flexibility in the code that can be used, and choice of transformation types to implement scanner defeats.

Other information, including register liveness and block-boundary, may also be needed, and these normally also require construction of a CFG.

One alternative strategy that can be used to ease the difficulty of generating the engine is to limit static analyses to relatively simple ones. This choice forces the engine to live with the potential impreciseness of the resullts. Clearly these will limit the Transfom organ since *conditional* transformations (see Section 4.4) may need to rely in precise information (such as “is the value n this register a constant?”) to be semantics-preserving.

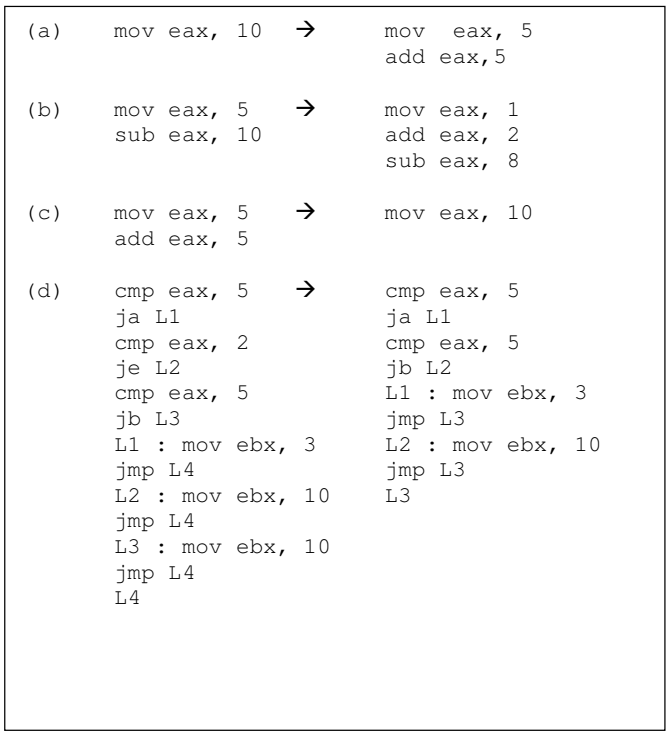
4.4 Transform

A simple choice to implement the Transform step is as a collection of transformation rules, such as those of a term-rewriting system (Walenstein, Mathur et al. 2006). When a code segment matches an instance of the left hand side (LHS) of a rule, it may be replaced by an instance of the right hand side (RHS) of the rule. The rule-based transformations can be further classified along two dimensions: a) the preconditions necessary for applying transformations and b) the transformation unit or the chunk-size of the LHS. Each dimension provides design options.

Preconditions for applying transformations.

The transformations can be classified into two categories: unconditional or conditional. The former type can be applied whenever a code segment matches the LHS of the rule; application of the latter transformation requires a match with the LHS *plus* certain other conditions must hold. Some of these conditions are exemplified by the rules in Figure 2 implemented in the virus W32.Evol. Rules (a) and (b) of this figure are unconditional rules, whereas the rules (c) and (d) are

conditional. The RHS of rule (c) modifies the register `eax`, which is not modified by the LHS. To ensure that the transformed program has the same behavior as the original program it is necessary



that the register `eax` not be live at the instruction where the rule is applied. The same condition also applies for (d), even though the LHS also modified `eax`. Nevertheless, the condition is not required for rule (b) in which eax is modified in the RHS, used, and then restored.

Figure 3: Single, block and SESE sub-graph transformations

Clearly, applying only unconditional transformations is easier for the metamorphic engine. However, this choice also makes it easy for an anti-malware scanner to reduce the various generations to a single form, provided the transformation rules are known (Walenstein, Mathur et al. 2006). To perform conditional transformations, the metamorphic engine must have the information required to verify the condition so that it can preserve the overall semantics of the program. In Figure 1 the task of computing this information is allocated to the Analyze organ.

Transformation chunk size.

To ensure correctness of transformations, it is ordinarily easiest if transformation rules are applied on single-entry single-exit (SESE) code segments (Lakhotia and Deprez 1998). Rules that are applied on SESE segments can further be classified on the chunk size of the SESE segment, i.e., whether it is a single instruction, a block of instructions, or a SESE subgraph. Figure 3 provides examples of these categories. Rule (a) transforms a single instruction; Rules (b) and (c) transform blocks of instructions; Rule (d) transforms a SESE subgraph. By definition, a single instruction is SESE, if exception conditions are ignored. So when the chunk size is a single instruction a rule may be applied if the LHS matches (and preconditions, if any, are satisfied). However, for chunk sizes greater than a single instruction it also becomes necessary for the metamorphic engine to ensure that the code segment matching the LHS is SESE, which in turn depends on correct construction of the CFG.

The choice of chunk size can significantly affect the type of transformations that can be performed in hopes of thwarting detection. W32.Evol only applies single-instruction transformations; as a

consequence its code monotonically increases over subsequent generations. On the other hand, Metaphor transforms single-instructions and blocks, thus causing its code to expand and shrink (Driller 2002). Though transformation of non-SESE segments is possible, the transformations also increase the theoretical challenges needed to ensure that the rules preserve correctness.

5. Conclusions

Metamorphism is known to have the potential to make detection of malicious code extremely difficult. While metamorphic malware to date have tended to be proof-of-concept and buggy, the threat of metamorphism is not to be taken lightly. We classify metamorphic malware as open-world vs closed-world and binary-transformer vs alternate-image transformer. Our analysis of the design space for closed-world binary transforming metamorphic malware yields an understanding of the space of design choices available to a malware author. Care was taken to link design choices available to the designer's goals and the implications and tradeoffs that the choices imply. The analysis makes it clear that the space is primarily structured by the needs of the metamorphic engine (its organs) and by the tradeoffs between transformation power and difficulty of implementation and analysis. The information needed to correctly implement the various transformations is a central theme.

This analysis of tradeoffs lays a foundation for anticipating future directions in metamorphic engine design. At this time, one possible conclusion that can be made is that in the near-term, the main analysis battlefield is likely to be over encoded or assumed properties. The tradeoff analysis provided suggests that as a reasonable conclusion because of a confluence of forces: encoding or assuming properties meets parsimony goals, it can force scanners to solve hard problems, and it frees the metamorphic malware designer to consider more powerful obfuscating or variation-inducing transformations that improve the ability to thwart detection. This then gives a malware author the upper hand and counters the earlier observation that “if a metamorphic malware can analyze itself, so can the scanner” (Lakhotia, Kapoor et al. 2005).

Further research is needed to improve the model of design space. One improvement that can be entertained is to create a more complete inventory of designer goals such that additional realistic design rationales can be constructed. A second advance would be to refocus the space on a more principled deconstruction of the information needed during transformation, and how it can be provided. This more detailed space is expected to be especially beneficial for predicting what properties are likely to be assumed, which information may be hidden and where, and how each choice influences the tug-of-war between the ‘vxers’ and ‘avers.’ It should also provide a more fine-grained vocabulary for discussing the classes of malware.

Acknowledgments

This work has been sponsored in part by funds from Louisiana Governor’s Information Technology Initiative.

References

Bruschi, D., L. Martignoni, et al. (2006). Detecting Self-Mutating Malware Using Control Flow Graph Matching. Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), IEEE Computer Society.

Bruschi, D., L. Martignoni, et al. (2006). Using Code Normalization for Fighting Self-Mutating Malware. Proceedings od the International Symposium of Secure Software Engineering, IEEE Computer Society.

Driller, M. (2002) Metamorphism in practice or "How I made MetaPHOR and what I've learnt". VX Heavens. http://vx.netlux.org/lib/vmd01.html.

Dube, T. E., K. S. Edge, et al. (2006). Metamorphism: A Software Protection Mechanism. International Conference on Inforfmation Warfare (ICIW2006), University of Maryland Eastern Shore, USA.

Edge, K.S., Dube, T.E., et al. (2006). A Taxonomy of Protection Used in Computer Viruses and Their Application to Software Protection, International Conference on Inforfmation Warfare (ICIW2006), University of Maryland Eastern Shore, USA.

Jha, C. M., S. Seshia, et al. (2005). Semantics-Aware Malware Detection. IEEE Symposium on Security and Privacy: 32-46.

Jordon, M. (2002). Dealing with metamorphism. Virus Bulletin**:** 4-6.

Kruegel, C., E. Kirda, et al. (2005). Polymorphic Worm Detection Using Structural Information of Executables. Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID), Springer-Verlag.

Lakhotia, A. and J.-C. Deprez (1998). Restructuring Programs by Tucking Statements into Functions. Journal of Information & Software Technology **40**: 677-689.

Lakhotia, A., A. Kapoor, et al. (2004). Are Metamorphic Viruses Really Invincible? - Part I. Virus Bulletin**:** 5-7.

Lakhotia, A., A. Kapoor, et al. (2005). Are Metamorphic Viruses Really Invincible? - Part II. Virus Bulletin**:** 9-12.

Lakhotia, A. and M. Mohammed (2004). Imposing Order on Program Statements and its implication to AV Scanner. Proceedings of 11th IEEE Working Conference on Reverse Engineering, Delft, The Netherlands, IEEE Computer Society Press, Los Alamitos, CA.

Linn, C. and S. Debray (2003). Obfuscation of Executable Code to Improve Resistance to Static Disassembly. Conference on Computer and Communications Security, Washington D.C., USA, ACM: Association for Computing Machinery.

Singh, P. K. (2002). A Physiological Decomposition of Virus and Worm Programs. Center for Advanced Computer Studies, University of Louisiana, Lafayette.

Stepan, A. E. (2005). Defeating Polymorphism: Beyond Emulation. Virus Bulletin Conference.

Ször, P. and P. Ferrie (2001). Hunting for metamorphic. Virus Bulletin Conference, Prague, Czech Republic.

Vecna (1998). Miss Lexotan 8, 29A E-Zine.

Walenstein, A., R. Mathur, et al. (2006). Normalizing Metamorphic Malware Using Term Rewriting. Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006).

Z0mbie. About reversing. VX Heavens. http://vx.netlux.org/lib/vzo14.html.

Z0mbie. Some ideas about metamorphism. VX Heavens. http://vx.netlux.org/lib/vzo20.html.