A Transformation-based Model of Malware Derivation

Andrew Walenstein
School of Computer Science and Informatics,
University of Louisiana at Lafayette, Lafayette, LA, USA
Email: walenste@ieee.org

Arun Lakhotia
Center for Advanced Computer Studies
University of Louisiana at Lafayette, Lafayette, LA, USA
Email: arun@louisiana.edu

***Abstract*—Since most malware is derived from prior code, understanding malware derivation and evolution is essential for many types of malware analysis. However prior models of malware relationships are insufficiently precise or fail to capture important relationships. A framework is proposed that treats both production and evolution uniformly as compositions of code transformations, and distinguishes disjoint but interleaved evolution of production code and malware code. Evolution relations are defined in terms of path patterns on derivation graphs; this generalizes and formalizes the relationship between phylogenies and provenance graphs. The comprehensiveness of the modeling framework is demonstrated using examples from the literature; implications for future work in relationship reconstruction are drawn.**

***Keywords*-malware, provenance, derivation, phylogeny, evolution, genome, polymorphism, attribution**

I. Introduction

There is a growing interest in recovering relationships between malware, often for the purposes of attribution. The attribution takes the form of developing a profile of the potential perpetrators, such as, their intent, capacity, training, geographical locations, targets, etc. For instance, in a recent study Symantec [1], after sifting through the data of malware over four years determined the same group was likely behind a host of attacks against defense, automotive, financial, and such companies. This group, that also targeted human rights organizations, was very sophisticated and was likely funded by a nation-state. The report goes on to map out the likely *modus operandi* of this purported group. In another study, Kaspersky [2] found that the developers of Stuxnet and Duqu were related in that they used the same attack platform called Tilded. This study also found that even though Flame uses a different architecture, "the team shared source code of at least one module in the early stages of development" proving that the teams had some contact.

It is not surprising that newly discovered malicious files turn out to be, almost invariably, some variation of previously known malware. In part, this is because true invention of entirely novel things is difficult, but is also due to the fact that attackers *reuse* and *modify prior code* in order to minimize their effort and production cost, and to also maximize speed of recovery from detection [3, 4]. Malware authors modify their code in some ways that are familiar to any software engineer: they can generate new *releases* such as by adding "features"—like password stealing—to an existing back door program; they can also create multiple *configurations* to enable running on different platforms. Other methods of change include recompiling [5], packing [6], permuting, obfuscating [7], or otherwise tweaking their programs [4].

The Symantec and Kaspersky studies, cited above, relied carefully on taking advantage of the reuse of code and expertise to thread together attacks that otherwise looked disparate. For instance, Kaspersky [8], states the following: "*The connection between Duqu and Stuxnet was revealed during the analysis of one of the incidents with regard to Duqu. During the investigation of the infected system thought to have been attacked in August 2011, a driver was found that was similar to the one used by one of the versions of Stuxnet. Though there were clear likenesses between the two drivers, there were also some differences in the details, such as the date of signing of the digital certificate. Other files which it was possible to attribute to the activity of Stuxnet were not found, but there were traces of activity of Duqu.*"

Even though the aforementioned studies relied on carefully studying similarities and differences between malware there has been little work in systematically and comprehensively modeling and capturing malware evolutionary relationships. The relationship models currently employed are *ad hoc*, designed by malware researchers on a case-by-case basis. Absence of a systematic and comprehensive model limits the sharing of experiences and creation of tools that use these models to support analysis, thereby limiting the capabilities for detecting, attributing, classifying, and naming malware.

Unfortunately the relationships between malicious files can be complicated, and have not been understood systematically and comprehensively. Two important current problems in capturing evolution relationship in malware are the lack of conceptual clarity and the lack of an appropriate and precise formalism to model the relationships.

Regarding conceptual clarity, sometimes ideas and terminology from *systematics* are adapted to speak in terms of "evolution" [10] of malware. Then malware "species" relations could be modeled in terms of a *phylogeny* [11] indicating

This material is based upon work supported by the Air Force Research Lab, the Defense Advanced Research Program Agency, and the Air Force Office of Scientific Research under Award No. FA8750-10-C-0171 and FA9550-09-1-0715.

978-1-4673-4879-9/12/$31.00 ©2012 IEEE

species inheritance relations. But does the idea of "species" map appropriately to malware? Are there reasonable analogues to *genotypes* and *phenotypes* [12]? Perhaps a species-level view does not fit what we know of malware production? We know that malware authors generally change their code in thoughtful, designed steps rather than through the repeated process of mutation and selection known to biology. So perhaps it is better to think of malware being related in a *family tree* (i.e., a parent–offspring graph), a *provenance graph* [13] or *software version model* [14] rather than a phylogeny? But if malware relationships are treated as relating only individuals, how do we identify points of evolution, and do we give up identifying groupings of individuals akin to "families" or "species"?

Regarding appropriate formalisms for modeling malware relationships, existing approaches fail to capture the known rich collection of relationships. Phylogenetic modeling approaches do not make explicit the individual relationships between executables grouped into species. For example, it does not seem reasonable to think that adding a junk byte to the end of an executable file [4] creates a new "species" of malware. Thus, a classic phylogenetic tree cannot represent this relation. Family trees, provenance graphs, and version models of programs typically permit fine-grained specification of relationships between program items, but lose the genetic view that permits separation of evolution (e.g., creating new programs) from mere derivation (e.g. compiling). Critically, models of either kind fail to formalize a critical fact of malware production: that some of the source involved is used to generate transformers, such as, compilers and obfuscators, which are then used to generate malware code or may be other transformers.

What is needed is a refined theory of malware evolution together with a principled, systematic framework for formalizing malware derivation relationships, and methods for expressing common relationships. Perhaps ideally the framework answers questions about what should be considered genetic material, and precisely defines the differences between familial relationships and species relationships. This paper proposes a new framework for modeling relationship that is designed to meet these requirements. Contributions arise primarily from three key propositions and insights underlying the framework:

- **Transformation-based model and formalism.** The framework treats all forms of malware relationship and derivation uniformly as compositions of code transformations. The transformations are decomposed into three classes (preserving, mutating, and combining). The models thus makes no bright-line distinction between automated, human-assisted, or wholly human code transformation. We call this the *transformation-based model of malware evolution.* This formalism permits a comprehensive system of specifying malware derivation relationships.
- **Disjoint genomes.** The framework identifies source code as genetic material, and proposes to identify two disjoint *genomes*, namely, the *production* and *malcode* genomes, and argues that these evolve independently. We call this the *disjoint genome assertion.* The framework identifies distinct lineages by distinguishing code that participates in the production of binaries (production code) from the code that is transformed into these binaries (malcode).
- **Evolution relation abstraction.** We propose that useful familial and classically-evolutionary relationships can be expressed as abstract patterns of composition on the fine-grained *derivation graphs*. We propose the use of *path expression on derivation graphs* to formalize common relationships found in malware, and provide specifications of common malware relationships of descent, sibling, and two types of polymorphic variant.

The framework is motivated through study of prior modeling approaches and analysis of published known malware relationships. Section II introduces the modeling problems through analysis of prior work, and a focused survey of known important malware relationships. Section III outlines a set of requirements for a malware relationship framework. The malware derivation model is proposed in Section IV and is evaluated informally in Section V by illustrating how the framework successfully models the motivating examples of Section II.

II. Malware Relationship Models

A model of malware relationship describes or explains how malicious files relate to one another through class or through derivation (copy, packed version, sibling, species, family, etc.). This section summarizes prior efforts in modeling malware relationships and their limitations.

A. Formal Models of Viral Sets

Some malware relationship models use mathematical formulations to define classes of malware. Cohen formalized related malware as "viral sets" using "Turing-like" machines [15]. Zuo *et. al* [16] and Bonfante *et. al* [17] used recursive function theory to define related malware through polymorphic generators and mutation. Filiol [18] defined related polymorphic and metamorphic in terms of the language generated by grammars.

These models primarily identify some classes of related malware, and do not explicitly identify evolutionary relationships apart from treating versions derived by the malware itself (as in polymorphic or metamorphic viral sets). For example, these models explicitly exclude the case where authors manually edit code to add features.

B. Provenance Models

A *provenance* model is a representation of how an artifact was constructed from materials through some process of transformation. Many different provenance models have been proposed for domains such as libraries and curators [22], distributed systems [23], security [13], workflows, and databases [24]. These are almost universally graphical models of data items and transformation processes that generate new products. Of particular interest are models of "toolchain" provenance for programs. Rosenblum *et. al* [3, 5]

TABLE I
COMPARISON OF KEY DEFINITIONS OF PRIOR EVOLUTION MODELS

	Inheritance			
Reference	Unit	Structure	Genome	Phylogeny instance
Goldberg *et. al* [11]	Byte strings	Multiple inheritance	DAG	Arborescence
Hayes *et. al* [19]	Functionality	Single inheritance	Lattice	N-ary tree
Dumitras *et. al* [20]	Function	Single inheritance	Unstated	Unstated
Erdelyi *et. al* [21]	CFG	Single inheritance	Unstated	Unstated

define a "hierarchical" model of provenance that permits them to establish relationships between executables and the compilers that generated them. Their model permits identification of the responsible compilers and their compiler settings, and permits mapping of specific segments of an executable to the compilers that produced them. Dumitras *et. al* [20] mention that compiler choice is a parameter of provenance they wish to consider, but do not propose an explicit relationship model. Note that software *version models* [14] can be considered as a type of provenance model, and so will share similar capabilities and limitations.

As with the formal models of viral sets, evolutionary relationships are not addressed by the provenance models, even those applied to security [13]. Neither Rosenblum *et. al* nor Dumitras *et. al* relate the malware instances to their ancestors or relatives. Another significant omission of the provenance models is that they do not model the case where an artifact is turned into a transformer. For example, the code of a program-to-program obfuscator is an artifact, and a compiler can transform that code to an executable that can be run to transform other artifacts (or perhaps even the obfuscator executable itself). The obfuscator code can evolve, meaning both provenance and evolution of obfuscating executables need to be tracked.

C. Malware Evolution Models

The idea that software evolves is not new. Lehman and others studied the evolution dynamics of software systems since the mid-1970s (e.g., Belady *et. al* [25]). Fred Cohen's dissertation [26] in 1985 likened computer viruses to forms of *artificial life* that are able to generate new descendants. Since then, many concepts of biological evolution have been commonly applied to malware.

In some cases, the language of evolution is used loosely to refer to general changes over time rather than the rich biological model of actual inheritance of code with variation. In other cases, the concepts from biology are more directly applied. For instance, genes are sometimes identified as source code [27], the defense-attack "arms race" has been cast as a form of co-evolution [28], and malware variations have been explained as polymorphisms or metamorphisms [10]. There have also been many different proposals from the anti-virus industry for naming of "species" [29], and separation of malware into different "types" ("bots", "viruses", "downloaders", etc.). These classes are derived by subject matter experts as being useful divisions, but they do not formalize the range of evolution relationships found. Other classification approaches utilize forms of machine learning and automated classification to build categories (see e.g., Shabtai *et. al* [30]), but in these the evolution between elements within and between categories is not explored.

More detailed and specific models of malware evolution can be found either explicitly in papers proposing methods for phylogeny reconstruction, or implicitly in papers recounting and analyzing specific histories of evolution.

Most examples of automated phylogeny reconstruction studies, such as those of Dumitras *et. al* [20], and Hayes *et. al* [19] are explicitly single inheritance, that is, they assume that source from different malware lineages could not have combined to form a new lineage. Of the reviewed models, most either assume the units of inheritance are the binary codes (the executables), or leave the true units of inheritance unstated. Some, such as Dumitras *et. al* [20] propose cladistic study based on *expressed features*, i.e., the functions that are in common for a group of related malware instances. Table I summarizes reviewed work.

Published accounts of specific instances of malware evolution (i.e., manual phylogenetic analysis) rarely go into significant detail about the evolution model (definition of genes, etc.), but rather implicitly assume a notion of *species* or *class*. Mathur *et. al* [31] and Schipka [4] are notable exceptions that recount characteristics and sources of regular patterns of change in observed variations. Table II provides a summary of the subjects and types of evolution models assumed in notable evolution accounts.

TABLE II
NOTABLE EVOLUTION HISTORY REPORTS

Reference	Subjects	Structure	Inheritance
Gordon [32]	Agobot, Sdbot	tree, list	source
Gordon [33]	Beagle		source
Canavan [34]	Agobot, Sdbot		source
Mathur *et. al* [31]	Storm/Tibs	tree	source, keys
Schipka [4]	Warezov	tree	source, binary

III. Requirements For Malware Relationship Frameworks

This section analyzes the review from Section II and generates a set of goals and requirements for malware relationship frameworks.

A. Features from Biological Systematics

We may expect that a useful model of evolution contains many of the same features as models in biology: the units of inheritance are described, and the mechanisms and methods for inheritance and gene change are identified. We could also expect clear definitions of *genome*, and state how inheritance and phylogenetic relationships are established. In addition, the analogues for *gene expression* and the translation from genotype to phenotype (roughly speaking, the things they generate) should be included in the model (cf. Feitelson *et. al* [12]). In short, a model of malware evolution will explain how the different malware instances we find come about in terms of evolution and derivation.

B. Scenarios of Derivation

All modeling efforts strive for an appropriate balance between completeness and expressiveness, simplicity, precision, and formality. In relating to prior work it is helpful to identify cases where models fail to accurately characterize the relationships and evolution that occurs. To that end, consider the following scenarios, all either recounting published accounts of observed malware evolution, or directly inspired by them. These scenarios provide insights into the requirements of a modeling framework, and will be consulted again during evaluation of the proposed framework.

1) Multiple, separate descendant lineages: Author "[sd]" releases a modular bot constructor called *Sdbot* that generates bot source code. The code is adapted, customized, and extended by bot authors who create derivatives [34]. Each of the various lines of derivatives share common code but also have their own distinctive features added by their authors.

2) Interleaved code sharing: The *Agobot* bot constructor is released in the form of a modular source base. Innovations are added to the Agobot code over time, and these are mixed (i.e., copied) with Sdbot code base such that variants could be found with large segments of code from both Agobot and Sdbot [34]. This mixing of code between lineages is reminiscent of "horizontal gene transfer" (HGT) in biology that makes it difficult to distinguish clear species in bacteria [35]. Indeed, HGT in bacteria is suggested as a mechanism for rapid spread of resistance to new drugs, which would be a great analogy to the spread of resistance to new defenses by malware through exploit sharing.

3) Dependent lineage update: Four versions of Sdbot are released; multiple bot developers create distinct variants from the code [34, 36]. In such cases, the transfer of code from one lineage to another is not a one-time event; rather "mutations" of source genes in the "original" lineage (Sdbot code itself) are propagated to dependent lineages.

4) Differences in generation mechanism: Different compilers and compiler settings are used to compile malware code [36]. The resulting executables are different from an identical source.

5) Variation after compilation: Thirteen bot downloaders are created by automated mutation of a single source code base and then compiling, and the resulting binary further mutated to create new variations [4]. The mutations are by relatively simple binary-to-binary code morphing that does not change the functionality of the code.

6) Shared generated code and characteristics: A particular black-market packer [37] is used by two malware authors that work otherwise completely independently. Because the packer injects randomization and specific obfuscations, each author creates multiple variations of a single executable, and these variations share similar anti-disassembly and anti-debugging functionality.

7) Shared functionality update.: Remote host exploit code in a worm is substituted with new "proof-of-concept" code published by a security researcher in a vulnerability disclosure.

8) Separate evolution of toolchain: The Tibs packer is used in the Storm malware to rapidly generate many variations from a single executable [31]. Over a period of months the author makes manual changes and implements a system that automatically generates additional changes to the packer outputs. During the same time, the "payload" executable undergoes evolution, effectively separating stealth evolution from malware functionality evolution.

These eight scenarios, in combination, inscribe an extensive space of malware evolution relationships. Fore example, the relationships between Duqu and Flame variants [2] may be related through types 1, 2, and 5.

Prior efforts in modeling such scenarios fail to capture the richness of the relationships between malware code. In some cases, malware families are directly related by source sharing, but may use different toolchains to generate the executables. Even the formal definitions that permit modeling of metamorphism does not describe the source-origin relations of the viral sets of polymorphically equivalent malware. These cases present problems for models that fail to capture multiple inheritance and sharing of code or libraries, or to differentiate between code the malware authors control versus code to generate the tools they use to generate binaries.

IV. Transformation-Based Evolution Model

A model of the derivation of malware code from collections of other code is presented. A key proposition in the model is that all derivation relationships can be uniformly modeled as a composition of various code-to-code transformations. The model distinguishes between *production* transformations and *evolution* transformations. Production transformations generate derived products from collections of sources. The end-result of production transformations is executable code. Evolution transformations generate new *source code* elements. In this model, therefore, genetic material is identified by inheritance after evolution.

A second key proposition is that there are two essentially disjoint lineages to distinguish as separate genomes: the *production* and *malcode* genomes. This proposition follows from the first by noting that the source code for the production transformations themselves may evolve, and that this evolution is independent from the evolution of the malware code being transformed in production. The independence is traditionally

maintained, as we argue in the following, because code maintainers normally fastidiously avoid hand editing artifacts derived through production transformations.

The model is formalized using sets and functions. Derivation graphs are introduced to model the compositions, and patterns of evolution are defined based on notable patterns of compositions.

A. Model Definition

1) Data sets: The following are basic definitions of the sets of values that the transformation functions operate on.

B. Let B be the set of *binaries.* This is the set of all executable codes for real machines. This is meant to include linked executables as well as compiled object files. Binaries may be loaded onto a virtual machine (such as an emulator), but there exists some hardware on which it runs natively.

V. Let V be the set of *virtual binaries*. This is the set of all executable codes for virtual machine hardware. This code must be emulated, compiled, or dynamically translated to native code.

H. Let H be the set of *higher-level code*. This is the set of all code not including V or B. This is meant to include assembler as well as traditional high level languages. The model does not distinguish between languages or stratify them.

C. Define C to be the set of *code* as $C = B \cup V \cup H$.

2) Transformations: Classes (i.e., sets) of functions from the universe of all functions are defined using function type signatures on the base set classes. Any transformation function is also labeled as *preserving*, *mutating*, or *combining*. Preserving transformations are semantics- or behavior-preserving transformations [7]. Mutating transformations are those that do not preserve behavior or semantics. Both mutating and preserving transformations are unary functions from code to code ($C \rightarrow C$). Combining transformations are binary functions that merge two sets of codes in some manner. In the following paragraphs, preserving, mutating, and combining transformation functions are labeled with a superscript P, M, or C, respectively. The labeling is not intended to formally add power to the transformation function definition, but rather is used to help identify evolution changes.

1. $H \rightarrow H$

- *Translators*^{P}. This class transforms code from one high level language into another.
- *Generators*^{P}. These take code in one programming language and generate code in another. The Agobot construction kit is a type of generator.
- *Editors*^{M}. These can be ordinary code maintainers and developers. We can treat all of their edits as code-to-code transformations. Here we are not thinking of their *editing programs* as the editors, but the programmers themselves. While some sequences of code changes are behavior or semantics preserving, we consider here ones that change behavior or semantics.
- *Mutators*^{M}. Mutators can make random, pseudo-random, or planned changes to code in an automated manner. This class of transformation does not, in general, preserve semantics or behavior.

2. $H \rightarrow B$

Notable examples are classic assemblers and compilers. In this model, a compiler implements a specific transformation function. So a compiler program, such as gcc, run with different options is treated as a different function. These transformations generally aim to be semantics-preserving transformers.

3. $H \rightarrow V$

These are the virtual machine analogues to $H \rightarrow B$.

4. $V \rightarrow V$

Examples include Java or .NET bytecode optimizers, which are ordinarily semantics-preserving.

5. $V \rightarrow B$

This class of transformer include compilers such as Java to binary bytecode compilers, as well as "binary translation" frameworks such as "just in time" compilation of Java virtual machines.

6. $B \rightarrow V$

This class of transformations turn binaries into virtual binaries. This type of transformation is sometimes done to support legacy code,

7. $B \rightarrow B$

Binary-to-binary program transformers are commonly used in malware in the form of *Packers*^{P} and *Encrypters*^{P}. Both of these generate new executables from provided executables. Similar to $H \rightarrow H$ translators, there are binary-only versions of Editors and Mutators for binary code.

8. $H \times H \rightarrow H$

- *Patchers*^{C}. These modify code in ways that can add, modify, or delete parts of code based on a specification of the update to provide.
- *Copy-pasters*^{C}. Ordinary software engineers frequently copy and paste code. Such plagiarism is common in the malware community also, and was observed in the Agobot-Sdbot interactions [34].

9. $B \times B \rightarrow B$

- *Linkers*^{C}. A linker is responsible for combining binary components into a new binary executable. This is the primary method for combining code with third party libraries at compile time.
- *Loaders*^{C}. Like a linker, a loader also combines binary components into a new binary executable, but at run-time and to create an in-memory program image. This is a primary method for combining third-party dynamically-loaded libraries. Note that certain malware can be rendered incapable of infection by providing patched (up-

dated with vulnerability removed) dynamically-loaded libraries in place of vulnerable ones.

10. $I(C) \rightarrow (C \rightarrow C)$

I is an interpretation function. It "executes" the code for a transformation function, and is modeled as a function from code to transformation functions. For example, the source code $h \in H$ for a C++ compiler is interpreted by I to generate a function that maps C++ code to binaries. The interpreter function is essential for formalizing evolution of code that produces other code. For simplicity we do not consider the details of how I works, we only make use of its function.

Obfuscators, source-to-source transformers that make the code less scrutable [7, 38, 39] while preserving the original semantics, can be modeled in this framework as any code-to-code transformation. Obfuscating transforms need not be source-to-source. For example, certain packers obfuscate by transforming native binaries into virtual binaries for a randomly-generated virtual machine [40]. Such transformers too can be naturally modeled in our framework.

In the following, we use our model to more formally define some terms that we have used intuitively up to this point.

- *Derivation.* A derivation is a composition of transformation functions that map one code to another. We denote a derivation from one code to another as $C \xrightarrow{\star} C$; that is, it denotes a composite function made by composing 0 or more transition functions. We also generalize the combination transformation so that bags of code elements can be transformed to derive another code; this is denoted as $C^n \rightarrow C$. In this framework, derivation need not lead to a binary.
- *Production.* A production is a derivation utilizing no mutating transformations and yielding a B or V.
- *Evolution.* An evolution is a class of derivation utilizing only mutating transformations and yielding code that is used to produce new variants of malware. Note that evolution derivations do not overlap with production derivations.
- *Source code.* Code elements that are never results of production transformations, and are subject evolution transformations. That is, these are code elements that are units of inheritance—they are the *genes* of malware.
- *Production code.* Source code that is used to create code transformation functions. That is, the source code is subject to I, and the result is subsequently used in a production derivation.
- *Malcode.* Source code that is used in productions.

B. Example Model

The framework developed in the previous section provides the basic mechanisms for identifying a useful class of relationships between code relating to malware. Each relationship is identified by a composition of transformations. For example, let:

- $bot_1, bot_2 \in H$ be two completely unrelated complete malcode source files,
- $gcc \in H$ be a source code of a C++ compiler such that $gcc' = I(gcc)$ and $gcc' : H \rightarrow B$,
- $tibs \in H$ be the C++ source for the Tibs packer such that $tibs' = I(tibs)$ and $tibs' : B \rightarrow B$,
- $changealg : H \rightarrow H$ be an edit for $tibs$ that modifies its polymorphic unpack algorithm,
- $bugfix : H \rightarrow H$ be an edit for gcc,
- $spamit : H \rightarrow H$ be an edit that adds a new spamming capability to bot_2.

Then the following are all compositions denoting productions of executables, that is, all generate executables in B:

$$\begin{aligned} bot_1' &= tibs'(gcc'(bot_1)) \\ bot_2' &= tibs'(gcc'(bot_2)) \\ bot_2'' &= gcc'(bot_2) \\ bot_2''' &= tibs'(gcc'(spamit(bot_2))) \\ bot_1'' &= I(changealg(tibs))(gcc'(bot_1)) \\ bot_1''' &= tibs'[I(bugfix(gcc))(bot_1)] \end{aligned}$$

then

(a) bot_1', bot_2', and bot_2''' share a common production code base ($tibs \cup gcc$). More specifically, they share the same production function $tibs' \circ gcc'$.

(b) bot_2' and bot_2'' share the same malcode base, but are generated by different productions.

(c) bot_2' and bot_2''' are malcode related by a mutation transformation (source code evolution). bot_2''' is related to bot_2' by mutation of malcode source bot_2 (products do not evolve).

(d) bot_1'' and bot_1''' are both related to bot_1' by evolution in production source ($tibs$ and gcc for bot_1'' and bot_1''', respectively).

These example relations show how production and evolution are unified in this transformation-based model, and illustrate different forms of malware relationship. To this point the various relationships described are not given names; this is done by using patterns defined on derivation graphs.

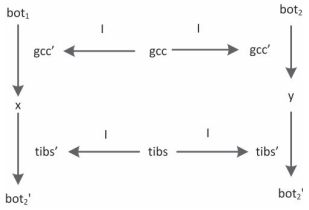
C. Derivation Graphs and Path Expressions

The notation for transformation composition is detailed and can be relatively confusing to read. Two alternative ways of presenting derivation information are informally defined.

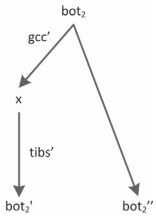
Definition 1: A *concrete derivation graph* renders elements in C as nodes, applications of elements from $C \rightarrow C$ as labeled arcs, and applications of elements from $C \times C \rightarrow C$ as nodes with two incident edges from elements of C, and one outgoing edge to an element of C.

Derivation graphs can be used to visualize portions of a derivation. Figures 1(a)–(c) visualize the four noted derivations from Section IV-B.

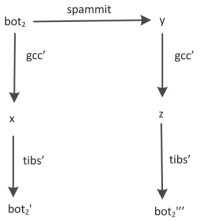
Concrete derivation graphs can model derivation steps at fine levels of granularity. However, frequently one wishes to model and consider only certain aspects of a relationship, only certain key steps within some derivation graph, or only



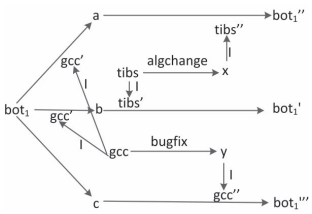
(a) common production code base



(b) common malcode base, different productions

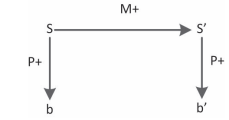


(c) malcode evolution, common production

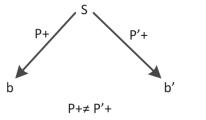


(d) production evolution, common malcode

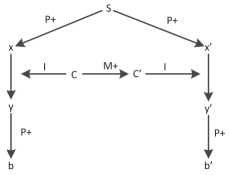
Fig. 1. Concrete derivation graphs corresponding to example from Section IV-B



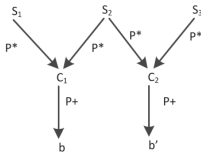
(a) descendant / ancestor



(b) polymorph



(c) e-polymorph



(d) sibling

Fig. 2. Some patterns of evolution relationships

certain classes of derivation. This can be accomplished with an abstraction.

Definition 2: An *abstract derivation graph* is a graph where nodes and edges are either ones from concrete graphs or are abstracted to generic classes or compositions. Specifically, nodes are labels representing specific code, one of the classes of code (H, B, V, or C) to denote arbitrary elements from those classes, a label representing a particular combining transform, or the letter C to denote arbitrary transform. Edges are labels representing specific transformations, regular expressions on the alphabet $\{P, M, X\}$ to represent compositions of transformation functions from preserving, mutating, and either preserving or combining classes of transformations, respectively.

This definition introduces generalizations for compositions of transformation functions using abstract classes and a simple form of path expression. They might reasonably be thought of as being opposite the *graph refinements* of hierarchical provenance models (see e.g., Moreau *et. al* [22]). The notation $C \xrightarrow{P} C$ and $C \xrightarrow{M} C$ indicate a single production and mutation composition, respectively, from C to C. Regular expressions are used to generalize the composition: $B \xrightarrow{P|M} B$ denotes either a production or mutation transformation on binaries, and $H \xrightarrow{M\star} H$ denotes a composition of zero or more mutations on high level code.

Abstract derivation graphs make it possible to specify particular relations between incidents of code. This makes it possible to precisely specify generalized evolutionary and production relationships. Figure 2 provides examples of common derivations. Figure 2(a) shows how a binary b can be an ancestor of b'. If the transition between s and s' were written $s \xrightarrow{M} s'$ the relation would be "parent", and if it were written $s \xrightarrow{MMM} s'$ or $s \xrightarrow{M\{3\}} s'$ it would be "great-grandparent". Figures 2(b) and 2(c) show how the framework concisely identifies two types of polymorphic variation, the second showing variation by evolution of production source code. These patterns can be composed to generate other classic familial relationships, such as "cousin".

V. Evaluation and Discussion

The modeling framework proposed in Section IV attempts to strike a balance in which important classes of malware relationships can be modeled without requiring onerous modeling formality or completeness. We informally evaluate the framework by considering how they can model the scenarios from Section III, and in terms of what implied statements the framework makes about the nature of malware evolution.

An anecdotal evidence of the value of this model comes from the conclusions of Gostev and Soukenkov's report on analysis of Stuxnet and Duqu [9], which states, "*From the data we have at our disposal, we can say with a fair degree of certainty that the "Tilded" platform was created around the end of 2007 or early 2008 before undergoing its most significant changes in summer/autumn 2010. Those changes were sparked by advances in code and the need to avoid*

detection by antivirus solutions. **There were a number of projects involving programs based on the "Tilded" platform throughout the period 2007-2011. Stuxnet and Duqu are two of them–there could have been others, which for now remain unknown.** *The platform continues to develop, which can only mean one thing–we're likely to see more modifications in the future.*"

The "Tilded" platform alluded to in the above report constitutes a production transformer. This report hypothesizes about the existence of such a platform based on similarities of obfuscation methods employed in Stuxnet and Duqu, two otherwise independent worms. The researchers infer that the production environment has undergone several revisions, and is likely to continue further revisions. Our framework provides a structure of modeling such relationships, both, of the evolution of the production transformations as well as the payload. Extracting information from malware to populate a derivation graph to draw inferences, such as those made in the above report, is a worthy research problem.

A. Comprehensiveness

The framework permits modeling at concrete, fine granularity of familial relationships (individual compositions and mutations) as well as abstract, large granularity such as phylogenetic relationships such as "common ancestor". We can consider how it addresses each of the limitations noted from prior modeling efforts:

- *Multiple, separate descendant lineages.* These can be modeled with a shared parental source code and separate mutation transformations for each of the leaves of the tree.
- *Interleaved code sharing.* The derivation graphs directly represent non-tree structures using combination transformations that can model copying of code to and from separately evolving lineages (a horizontal gene transfer analogue).
- *Dependent lineage update* and *shared functionality update*. These can be represented by a "trunk" of mutations of a source lineage that branch out to be combined at different points of parallel "trunks" of lineages that utilize these updates.
- *Differences in generation mechanisms* and *variation after compilation.* Both of these are instances of different production paths from a malcode source.
- *Shared generated code and characteristics.* This is modeled using a production code source that is shared through I (interpretation) applications for different malware lineages.
- *Separate evolution of toolchain.* The separation of production and malcode sources and their evolution histories make it possible to trace each independently.

The derivation graphs in Figures 1 and 2 illustrate these.

B. Limitations

For simplicity, the model's definition of source code specifically excludes machine generated code. In some cases this will be overly restrictive since it is quite possible that malware authors begin with a generated template or initial code base and then hand edit it from that initial phase. However in practice developers are generally loath to hand modify machine-derived code, such as parser-generator outputs, obfuscated code, or binary outputs. Binaries modified using automated transformation tools, can however be represented in the model, as also transformations on source code using tools, such as, automated indenters, formatters, and semantics-preserving refactoring tools.

The separation of malcode from production code is useful but artificial. Assuming it technically precludes modeling the case where benign code is being compiled with a compromised compiler that emits malicious code [41]. However the transition-based composition framework still permits it to be modeled, but the definition of "malcode" needs to be modified appropriately.

Besides relating malware samples using shared code, obfuscations, cryptographic algorithms, and packers, as done in the Symantec [1] and Kaspersky [2] studies, malware samples may also be related by their behavior. For instance, two samples connecting to the same IP address or a website may be considered related. Or malware samples may be related by relation between the sites, for instance, to determine that an attack is a "watering hole attack" [1]. Such relations to objects outside the code and production environment are not directly represented in the model.

VI. Conclusions

Government and industry is increasingly interested in attributing malware to its potential developers. This is done, in part, by relating disparate pieces of malware to develop a profile of the attacker. Prior efforts at characterizing relationships between instances of malicious code have not yielded a cumulative and comprehensive overview of the variety of relationships found in the real world malware, and is not suitable for use for attribution. The proposed framework captures and summarizes a wide variety of these relationships, and introduces an approach for modeling these at different levels of specificity. If desired, the fine-grained provenance and derivation history can be modeled to characterize the relationships between individuals in an analogy to a "family graph". But also, one may sketch out patterns of derivation—siblings, grandparents, and so on—by abstracting family graph nodes and edges to derivation chains and source code classes. With this new tool for modeling we expect the framework to contribute positively to reporting and comparison of malware evolution, and to enable more precise descriptions.

In addition to systematizing past knowledge and unifying it in a single framework, analysis of the framework identifies many possible avenues for future research. Relationships between malware binaries are complex: production is often far more complicated than simply determining compilers used; with a thriving black market and code sharing common, executables are frequently derived from a stew of source ingredients. This analysis suggests that existing work has in some

cases only scratched the surface of the problem of recovering provenance and derivation relationships (i.e., phylogenies), classifying malware into related groups (i.e., "families"), and reconstructing lineages between different sources.

REFERENCES

[1] O'Gorman, G., McDonald, G.: The Elderwood Project (August 2012)

[2] Kaspersky Lab: Resource 207: Kaspersky Lab Research proves that Stuxnet and Flame developers are connected (2012) Last accessed: Sept 13, 2012.

[3] Rosenblum, N.: The Provenance Hierarchy of Computer Programs. PhD thesis, University of Wisconsin-Madison, Computer Science Department (2011)

[4] Schipka, M.: A road to big money: evolution of automation methods in malware development. In Martin, H., ed.: Proceedings of Virus Bulletin Conference 2007, Vienna, Austria, Virus Bulletin Inc. (2007)

[5] Rosenblum, N., Miller, B.P., Zhu, X.: Recovering the toolchain provenance of binary code. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA'11), New York, NY, USA, ACM (2011) 100–110

[6] Xu, C., Andersen, J., Mao, Z.M., Bailey, M., Nazario, J.: Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In: Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN 2008). (June 2008) 177–186

[7] Collberg, C., Nagra, J.: Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Addison Wesley (2010)

[8] Kaspersky Lab: Kaspersky Lab Experts: Duqu and Stuxnet not the only malicious programs created by the responsible team (December 2011) Last accessed: Sept 13, 2012.

[9] Gostev, A., Soumenkov, I.: Stuxnet/Duqu: The evolution of drivers (December 2011)

[10] Ször, P., Ferrie, P.: Hunting for metamorphic. In: Proceedings of the 11th International Virus Bulletin, Prague, Czech Republic, Virus Bulletin (2001) 123–144

[11] Goldberg, L., Goldberg, P., Phillips, C., Sorkin, G.: Constructing computer virus phylogenies. Journal of Algorithms **26**(1) (1998) 188–208

[12] Feitelson, G., Treinin, M.: The blueprint for life? IEEE Computer **35**(7) (2002) 34–40

[13] Cheney, J.: A formal framework for provenance security. In: Proceedings of the 24th IEEE Computer Security Foundations Symposium, Cernay-la-Ville, France, IEEE Computer Society (June 2011) 281–293

[14] Conradi, R., Westfechtel, B.: Version models for software configuration management. ACM Computer Surveys **30**(2) (June 1998) 232–282

[15] Cohen, F.: A formal definition of computer worms and some related results. Computers and Security **11**(7) (1992) 641–652

[16] Zuo, Z., Zhou, M.: Some further theoretical results about computer viruses. The Computer Journal **47**(6) (2004) 627–633

[17] Bonfante, G., Kaczmarek, M., Marion, J.Y.: On abstract computer virology: from a recursion-theoretic perspective. Journal in Computer Virology **1**(3-4) (2006) 45–54

[18] Filiol, E.: Metamorphism, formal grammars and undecidable code mutation. International Journal of Computer Science **2**(1) (April 2007) 70–75

[19] Hayes, M., Walenstein, A., Lakhotia, A.: Evaluation of malware phylogeny modelling systems using automated variant generation. Journal in Computer Virology **5**(4) (2009) 335–343

[20] Dumitras, T., Neamtiu, I.: Experimental challenges in cyber security: a story of provenance and lineage for malware. In: Proceedings of the 4th conference on Cyber Security Experimentation and Test Conference (CSEET'11), USENIX Association (2011)

[21] Erdélyi, G., Carrera, E.: Digital genome mapping: advanced binary malware analysis. In Martin, H., ed.: Proceedings of the 15th Virus Bulletin International Conference, Chicago, IL, U.S.A., Virus Bulletin Ltd. (2004) 187–197

[22] Moreau, L., Freire, J., Futrelle, J., McGrath, R., Myers, J., Paulson, P.: The open provenance model: An overview. In Freire, J., Koop, D., Moreau, L., eds.: Provenance and Annotation of Data and Processes. Volume 5272 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2008) 323–326

[23] Souilah, I., Francalanza, A., Sassone, V.: A formal model of provenance in distributed systems. In: First workshop on on Theory and practice of provenance. TAPP'09, Berkeley, CA, USA, USENIX Association (2009) 1:1–1:11

[24] Glavic, B., Dittrich, K.: Data provenance: A categorization of existing approaches. In: Proceedings of BTW 2007: Business, Technology und Web. 227–242

[25] Belady, L.A., Lehman, M.M.: Evolution dynamics of large programs. IBM Systems J. **15**(3) (1976)

[26] Cohen, F.: Computer Viruses. PhD thesis, University of Southern California (1985)

[27] Apagow, P.M.: Computer viruses: the inevitability of evolution? In Green, D.G., ed.: Complex Systems: From Biology to Computation. IOS Press (1992) 46–54

[28] Nachenberg, C.: Computer virus-antivirus coevolution. Communications of the ACM **50**(1) (1997) 46–51

[29] Leitold, F.: The solution in the naming chaos. In Turner, P., Broucek, V., eds.: EICAR 2005 Conference: Best Paper Proceedings. (2005) 365–378

[30] Shabtai, A., Moskovitch, R., Elovici, Y., Glezer, C.: Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey. Information Security Technical Report **14**(1) (2009) 16 – 29

[31] Mathur, R., Kapoor, A.: Exploring the evolutionary patterns of Tibs-packed executables. Virus Bulletin (December 2007) 6–9

[32] Gordon, J.: Agobot and the 'kit'chen sink Retrieved from `www.infectionvectors.com`, Feb 17, 2008.

[33] Gordon, J.: Year of the beagle: Parts I, II & III +supplement (July 2004) Retrieved from `infectionvectors.com`, Nov 10, 2011.

[34] Canavan, J.: The evolution of malicious IRC botnets. In: Proceedings of the Virus Bulletin Conference, Dublin, Ireland (2005)

[35] de la Cruz, F., Davies, J.: Horizontal gene transfer and the origin of species: lessons from bacteria. Trends in Microbiology **8**(3) (March 2000) 128–133

[36] Porst, S.: Comparing different versions of SDBot using SABRE BinDiff v1.7 (September 2005) Retrieved from `www.the-interweb.com` Nov 14, 2011.

[37] Chu, B., Holt, T.J., Joon Ahn, G.: Examining the creation, distribution, and function of malware on-line. Technical Report 23011, U.S. National Institute of Justice (March 2010)

[38] Giacobazzi, R.: Hiding information in completeness holes: New perspectives in code obfuscation and watermarking. In: Proceedings of the Sixth IEEE International Conference on Software Engineering and Formal Methods (SEFM'08), Cape Town, South Africa (2008) 7–18

[39] Dalla Preda, M.: Code Obfuscation and Malware Detection by Abstract Interpretation. PhD thesis, Dipartimento di Informatica, University of Verona (February 2007)

[40] Guo, F., Ferrie, P., Chiueh, T.: A study of the packer problem and its solutions. In Lippman, R., Kirda, E., Trachtenberg, A., eds.: Recent Advances in Intrusion Detection. Volume 5230 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2008) 98–115

[41] Thompson, K.: Reflections on trusting trust. Commun. ACM **27** (August 1984) 761–763