Using algebraic models of programs for detecting metamorphic malwares

N. N. Kuzjurin, , R. I. Podlovchenko, , V. S. Scherbina, , V. A. Zakharov

Abstract. Polymorphic and metamorphic viruses are the most sophisticated malicious programs that give a lot of trouble to virus scanners. Each time when these viruses infect new executables or replicate themselves they completely modify (obfuscate) their signature to avoid being detected. This contrivance poses a serious threat to anti-virus software which relies on classical virus-detection techniques: such viruses do not have any stable specific sequence of instructions to be looked for. In the ultimate case the only characteristic which remains invariable for all generations of the same virus is their functionality (semantics). To all appearance, the only way to detect for sure a metamorphic malicious code is to look for a pattern which has the same semantics as (i.e. equivalent to) some representative sample of the virus. Thus, metamorphic virus detection is closely related to the equivalence-checking problem for programs. In this paper we outline some new automata-theoretic framework for the designing of virus detectors. Our approach is based on the equivalence-checking techniques in algebraic models of sequential programs. An algebraic model of programs is an abstract model of computation where programs are viewed as finite automata operating on Kripke structures. Models of this kind make it possible to focus on those properties of program instructions that are widely used in obfuscating transformations. We give a survey (including the latest results) on the complexity of equivalence-checking problem in various algebraic models of programs and estimate thus a resilience of some obfuscating transformation commonly employed by metamorphic viruses.

1. Preliminaries

One of the most important branch in computer security is the designing of anti-virus software. According to current concepts [23], a *computer virus* is a self-replicating computer program which spreads by attaching copies of itself to programs and/or documents. But very often the term “computer virus" is referred to many other sorts of malware, such as worms, trojan horses, spyware, etc. The first anti-virus programs emerged in 1982 almost immediately with the advent of computer viruses [32]. Since that time we became the witnesses (and sometimes participants) of the unceasing “arm race" malware designers and anti-virus software engineers are involved in. Anti-virus programs attempt to identify, thwart and

eliminate computer viruses. Clearly, the crucial point is to detect a virus, and a number of techniques may be used to accomplish this task [22].

The suspicious behavior approach monitors the behavior of all programs. If one program displays a suspicious behaviour (say, tries to write data to an executable), the anti-virus software can pin-point such anomalies and alert a user. Unfortunately, modern nonmalicious programs offer many properties that are typical for viruses, and the boundary between legal and harmful behaviour is swiftly dispersing. Therefore the suspicious behavior approach suffers from the growing number of false positives and most modern anti-virus software uses this technique less and less.

A sandbox approach emulates the operating system, runs executables in this simulation and analyzes the sandbox for any changes which might indicate a virus. This approach is resource consuming and normally it takes place only during on-demand scans or in combination with the suspicious behaviour approach.

The most effective virus detection technique is based on virus dictionary. An anti-virus program tries to find virus-patterns inside ordinary programs by scanning them for so-called *virus signatures*. A signature is a characteristic byte-pattern that is a part of a certain virus or family of viruses. If a virus scanner finds such a pattern in a file, it notifies the user that the file is infected. When searching for virus signatures a virus scanner refers to a dictionary of known viruses that the authors of the anti-virus software have identified. Thus, virus detection problem is reduced to the well-known pattern-matching problem which can be solved by a vast amount of efficient algorithms.

The virus dictionary technique is effective only if virus signature remains immutable for all generations of the same virus. Therefore, in order to avoid detection some viruses attempt to hide their signatures. Polymorphic code was the first technique that posed a serious threat to virus scanners. *Polymorphic viruses* (zombie-6.b, f0sf0r0, Hare) use multiple techniques to prevent signature matching. First, the virus code is encrypted with a different key for each replica, and only a small routine remains in-clear to decrypt the code before running the virus. Second, when polymorphic virus replicates itself, it not only encrypts its body with a newly-generated key, but also applies *obfuscating transformations* [7, 8] to modify the decryption routine. A well-written polymorphic virus has no parts that stay the same on each infection, making it impossible to detect a virus directly by its signature. Anti-virus software deals with polymorphic viruses by performing heuristic analysis of the code and emulating the program in a sandbox to catch the virus when it decrypts its body in the main memory [24, 25].

Metamorphic viruses (Sobig, Beagle, Smile) attempt to evade heuristic detection techniques by using more complex obfuscating transformations and rewriting their bodies completely each time they are to infect new executables. As virus designers employ more complex obfuscating transformations in metamorphic engines, heuristic virus-detection techniques are bound to fail [34]. Metamorphic viruses also attempt to "weave" their code into the host program making detection by traditional heuristics almost impossible since the virus code is mixed with a program code [18].

Thus, polymorphic and metamorphic viruses throw down a serious challenge to virus scanners. The key issue which makes these viruses strongly resilient against conventional virus detection techniques is the using of obfuscating transformations. An obfuscating transformation $\mathcal{O}$ is any semantic-preserving transformation that brings a program π into such a form $\mathcal{O}(\pi)$ which is far less understandable than the original program π. Initially, obfuscation was intended to provide the protection of intellectual property on computer software [7, 8]. But subsequent investigations [1] have shown that obfuscation may be used for various applications including the designing of public-key cryptosystems, protection of "watermarkings" and mobile agents, and (last but not least) for writing stealth viruses as well. It was proved (see [1]) that a "black-box" secure obfuscation of all programs is impossible, though it can be achieved for some specific families of programs [35]. Nevertheless, even weak obfuscating transformations [3, 17] may obstruct program analysis algorithms. Experiments [5, 6] demonstrated that three commercial virus scanners widely used in common practice could be subverted by very simple obfuscating transformations (code transposition and `nop`-insertion). This means that further improvements of virus scanners to make them effective protection tools against stealthy viruses require some fundamental research on the potency of obfuscation and deobfuscation techniques.

It should be noticed that obfuscating transformations change the syntax of a program but preserve its semantics. We may assume that in the ultimate case the only characteristic which remains invariable for all generations of the same metamorphic virus is its semantics. Hence, the semantics of a virus as its true signature which is shared by all mutations of the virus code. Therefore, to check if a program Π is infected with a metamorphic virus π it is sufficient to find out whether Π contains a piece of code which has the same semantics as (or, in other words, is functionally equivalent to) π. Unfortunately, as it was demonstrated in [2], the virus detection problem is undecidable in general. But if we further assume that a transformation $\mathcal{O}$ employed for the obfuscation of the virus π is also known to anti-virus software engineers then the situation changes drastically. The point is that in practice the

obfuscating transformation $\mathcal{O}$ preserves an equivalence relation on programs $\sim_{\mathcal{O}}$ which is much stronger than the functional equivalence. Therefore, it is quite possible to capture the equivalence $\sim_{\mathcal{O}}$ within some rather simple model of computation (program model) and then develop an equivalence-checking technique $EqCheck_{\mathcal{O}}$ within this model. The equivalence-checking algorithm $EqCheck_{\mathcal{O}}$ may be used as a basis for constructing an anti-virus program which could carry out signature-matching modulo $\sim_{\mathcal{O}}$.

This approach to the metamorphic virus detection was initiated by Christodorescu and Jha [4]. They presented an architecture for detecting malicious pattern in executables that can cope with some simple obfuscating transformations, such as changes in control flow, register reassignments, and dead code insertion. Experimental results which demonstrate the efficiency of their prototype tool look highly encouraging. But at the same time the authors of the papers [5, 6] noticed that more advanced obfuscating transformations (e.g. code transposition) require further investigations. Some of these transformations essentially depend on algebraic properties of typical program instructions. Most of such properties can be specified in the framework of the theory of algebraic models of sequential programs developed in the series of papers [16, 20, 27, 28, 37, 40].

Algebraic models of programs approximate operational semantics of real imperative programs and facilitate thus the development of equivalence-checking algorithms and optimization transformations for programs. Given two sets of basic program instructions $\mathcal{A}$ and basic predicates $\mathcal{P}$, a program π in the framework of an algebraic model is viewed as a deterministic finite state automaton whose input alphabet is $\mathcal{A}$ and output alphabet the set $\rho(\mathcal{P})$ of all possible evaluation of basic predicates. Such an automaton $\mathcal{A}$ operates on a set $\mathcal{M}$ of labeled Kripke structures that provide a semantics of program instructions and predicates. The models of this kind are called algebraic since in the most practical cases the set $\mathcal{M}$ corresponds to some semigroup which captures the most important algebraic features of operational semantics of real programs. The main advantage of these models is their scalability: choosing a set of algebraic and logical properties of program instructions and predicates one may build an appropriate model of programs whose semantics captures exactly the selected set of properties. This is very much convenient for developing metamorphic virus scanners. Given an obfuscating transformation $\mathcal{O}$ employed by the metamorphic engine of the virus one could

- choose an algebraic model $\mathcal{M}$ such that $\pi \sim_{\mathcal{M}} \mathcal{O}(\pi)$ holds for every program π; in this case it is said that the obfuscation $\mathcal{O}$ is "visible" within the model $\mathcal{M}$;

- estimate the complexity of the equivalence-checking problem for programs in the model $\mathcal{M}$; this gives a rough estimate to the complexity of the virus detection problem;
- develop (if possible) an equivalence-checking algorithm for the model $\mathcal{M}$ and use it as a scanning tool for those metamorphic viruses that use the obfuscation $\mathcal{O}$.

The very concept of an algebraic model of programs was introduced by Ianov in his seminal paper [16]. Since that time the equivalence-checking problem for programs in algebraic models has been studied extensively and some uniform approaches that combine automata-theoretic and group-theoretic techniques were developed. It turns out (see [15, 29, 38, 40]) that for many algebraic models of programs the equivalence-checking problem is decidable in polynomial time. This causes us to anticipate that some metamorphic viruses could be detected uniformly in reasonable time through the advanced signature-matching technique. The similar approach which relies on abstract interpretation concept instead of algebraic models of programs was developed in [9, 10, 11] for evaluating the resilience of program watermarkings.

On the other hand, some latest results show that the equivalence-checking problem in some natural algebraic models is intractable. This means that obfuscating transformations in the framework of such models may have rather strong resilience against equivalence-checking detection machinery. We hope that further investigations will clarify this effect.

The rest of the paper is organized as follows. In Section 2 we introduce formally algebraic models of programs. In Section 3 and 4 we survey the complexity results to the equivalence-checking problem for some algebraic models of programs. Finally, in Section 5 we discuss the impact of these results on the metamorphic virus detection problem.

2. Algebraic models of programs

Algebraic models of programs deals with sequential computer programs at the propositional abstraction level. In this section we define the syntax and the semantics of propositional sequential programs (PSPs).

Let $\mathcal{A} = \{a_1, \ldots, a_r\}$ and $\mathcal{P} = \{c_1, \ldots, c_k\}$ be two finite alphabets. In what follows r and k denote the cardinality of the alphabets $\mathcal{A}$ and $\mathcal{P}$ respectively. We emphasize that these parameters are fixed for every algebraic model of programs to be considered. The elements of $\mathcal{A}$ are called *basic instructions*. Intuitively, each basic instruction stands for some assignment statement in imperative program. But rather well it may corresponds to a library function call, basic block, or any other fragment of a program which always terminates.

The elements of $\mathcal{P}$ are called *basic predicates*. They stand for elementary built-in relations on program data. Each basic predicate may be evaluated by $false$ or $true$. We write $\rho(\mathcal{P})$ for the powerset of $\mathcal{P}$; the elements from $\rho(\mathcal{P})$ represent all possible evaluations of basic predicates.

Definition 1. *A* propositional sequential program *(*PSP*, for short) is a finite transition system* $\pi = \langle V, \mathbf{entry}, \mathbf{exit}, T, B\rangle$*, where*

- V *is a non-empty set of* program points*;*
- **entry** *is the* initial point *of the program;*
- **exit** *is the* terminal point *of the program;*
- $T\colon (V - \{\mathbf{exit}\}) \times \rho(\mathcal{P}) \to V$ *is a (total)* transition function*;*
- $B\colon (V - \{\mathbf{exit}\}) \to \mathcal{A}$ *is a (total)* binding function.

A transition function represents the control flow of a program, whereas a binding function associates with each point some basic instruction. One may also consider a PSP as a deterministic finite state automaton (DFAs) operating over the input alphabet $\rho(\mathcal{P})$ and the output alphabet $\mathcal{A}$. By the *size* $|\pi|$ of a program π we mean the cardinality of the set V (assuming that the cardinalities of $\mathcal{A}$ and $\mathcal{P}$ are fixed).

The semantics of PSPs is defined with the help of semigroup Kripke structures.

Definition 2. *A* semigroup Kripke structure *is a triple* $M = \langle S, \circ, \xi\rangle$*, where*

- $(S, \circ)$ *is a* semigroup *generated by the set* $\mathcal{A}$ *of basic instructions, and;*
- $\xi : S \to \rho(\mathcal{P})$ *is a (total)* evaluation function.

A semigroup $(S, \circ)$ gives interpretation to the basic instructions. The elements of S may be viewed as data states, and the neutral element (unit) ε stands for the initial data state. When an instruction a, $a \in \mathcal{A}$, is applied to a data state s, $s \in S$, the result of execution of a is the data state $s' = s \circ a$. An evaluation function ξ is used for the interpretation of basic predicates: given a data state s, an evaluation $\xi(s)$ returns a set of all basic predicates that are evaluated to $true$ on s.

Let π be a PSP, and M be a semigroup Kripke structure. The *run* of π on M is a sequence (finite or infinite) of pairs

$$r(\pi, M) = (v_0, s_0), (v_2, s_2), \ldots, (v_i, s_i), (v_{i+1}, s_{i+1}), \ldots,$$

such that

1. $v_0 = \mathbf{entry}$, $s_0 = \varepsilon$ is the initial data state of M;
2. $v_{i+1} = T(v_i, \xi(s_i))$, $s_{i+1} = s_i \circ B(v_i)$ hold for every $i,\ i \geq 1$;
3. the sequence $r(\pi, M)$ either is infinite (in this case we say that the run *loops* and yields no results), or ends with a pair (v_n, s_n) such that $v_n = \mathbf{exit}$ (in this case we say that the run *terminates* with a result s_n).

We denote by $[r(\pi, M)]$ the result of a run $r(\pi, M)$ assuming that the result is undefined when $r(\pi, M)$ loops.

By an *algebraic model of programs* $\mathcal{M}$ we mean the set of all PSPs over fixed alphabets $\mathcal{A}, \mathcal{P}$ whose semantics is specified by the set $\mathcal{M}$ of semigroup Kripke structures.

Definition 3. *Given an algebraic model of programs $\mathcal{M}$, PSPs π_1 and π_2 are said to be* equivalent on $\mathcal{M}$ *($\pi_1 \sim_{\mathcal{M}} \pi_2$ in symbols) iff $[r(\pi_1, M)] = [r(\pi_2, M)]$ holds for every semigroup Kripke structure M from $\mathcal{M}$.*

The equivalence-checking problem (ECP for short) in an algebraic model of programs $\mathcal{M}$ is that of checking, given an arbitrary pair of PSPs π_1 and π_2, whether $\pi_1 \sim_{\mathcal{M}} \pi_2$ holds.

As it may be seen from the definitions above, an algebraic model of PSPs is just an abstract model of computation where programs are regarded as DFAs operating on Kripke structures. In this connection ECP in algebraic models of PSPs is nothing else but a generalization of the well-known equivalence-checking problem for DFAs.

We say that an algebraic model $\mathcal{M}$ *approximates* an equivalence relation $\equiv$ on the set of PSPs if $\pi_1 \sim_{\mathcal{M}} \pi_2$ implies $\pi_1 \equiv \pi_2$ for every pair of PSPs π_1 and π_2. Approximation relation constitutes a lattice on the set of all algebraic models of programs [27, 37]. Hence, given an equivalent (obfuscating) transformation $\mathcal{O}$ of programs, one may consider an algebraic model $\mathcal{M}_{\mathcal{O}}$ which is the best approximation to the equivalence relation $\equiv_{\mathcal{O}}$ induced by the transformation $\mathcal{O}$, and then analyze ECP for $\mathcal{M}_{\mathcal{O}}$. If the problem is undecidable (or intractable) then one may select a suitable approximation $\mathcal{M}'$ to the model $\mathcal{M}_{\mathcal{O}}$ and develop an efficient equivalence-checking algorithm for $\mathcal{M}'$. This algorithm may be employed by deobfuscating virus scanner for detecting those metamorphic malware whose obfuscation is achieved through the transformation $\mathcal{O}$.

In the next section we consider in more detail a number of algebraic models of programs that may be employed to uncover some common obfuscations.

3. Equivalence-checking problem for basic algebraic models of programs

We restrict our attention to the algebraic models of programs $\mathcal{M}$ whose structures $M,\ M \in \mathcal{M}$ are based on the same semigroup $(S, \circ)$. Every model $\mathcal{M}$ of this kind is completely specified by the pair (S, L), where $L = \{\xi : \langle S, \circ, \xi\rangle \in \mathcal{M}\}$ is the set of admissible evaluation functions. If, moreover, every evaluation function ξ defined on S is admissible (i.e. $\langle S, \circ, \xi\rangle \in \mathcal{M}$ holds for any ξ) then we will write (S, L_S) to specify such a model $\mathcal{M}$.

Next we consider several families of algebraic models of programs based on most simple semigroups. For every model $\mathcal{M}$ we discuss which type of obfuscation is "visible" within this model and estimate the complexity of ECP for $\mathcal{M}$. One may regard these estimates as some qualitative characteristics of resilience for those obfuscating transformations that are "visible" in the framework of $\mathcal{M}$.

3.1. Free semigroups. The maximal model $\mathcal{M}_0 = (S_0, L_{S_0})$.

The maximal model $\mathcal{M}_0$ is associated with the free semigroup S_0 generated by the set of basic instructions $\mathcal{A}$. This model is called maximal since it approximates any other algebraic model of programs. Obfuscating transformations in the framework of $\mathcal{M}_0$ may change the structure of control flow of a program but they can't affect the order of instructions execution. The maximal model was introduced by Ianov [16] in 1958 and soon received the name Ianov's schemes. In [16] it was proved that ECP for the maximal algebraic model of programs is decidable. In fact, this is the first positive result which demonstrated the applicability of formal methods for program analysis. Later in [31] it was shown that ECP for $\mathcal{M}_0$ is inter-reducible with ECP for DFAs. Taking into account the most efficient equivalence-checking algorithm for DFAs presented in [15] we arrive at

Theorem 1. *ECP "$\pi_1 \sim_{\mathcal{M}_0} \pi_2$?" in the maximal model $\mathcal{M}_0$ is decidable within a time $O(n \log n)$, where n is the total size of PSPs π_1 and π_2.*

There are many ways of refining the maximal model. For example, it may be assumed that a semigroup S_0 contains several unit elements. Every such unit element corresponds to a nonessential instructions whose execution does not change data states (like `nop` instruction in IA-32 instruction set). It is easy to verify that the complexity of ECP for this algebraic model is the same as for $\mathcal{M}_0$. This fact explains the effectiveness of a static analyzer for detecting malicious patterns in executables [4, 6] developed by Christodorescu *et al.*: it uncovers exactly those obfuscating transformations that are "visible" in the refined variant of the maximal

algebraic model of programs.

Other refinements of $\mathcal{M}_0$ are also of some interest for program obfuscation. Denote by $mod(a)$ the set of program variables whose values are modified by an instruction a, and by $used(a)$ ($used(p)$) the set of program variables whose values are used by an instruction a (predicate p). Clearly, if $mod(a) \cap used(p) = \emptyset$ then the truth value of p before and after the execution of a is the same. In this case it is said that the predicate p is *invariant* w.r.t. the instruction a. For example, the predicate `y==0` is invariant w.r.t. the instruction `x++`. Usually, a very simple syntactic analysis is able to check whether some predicate p is invariant w.r.t. a instruction a. Algebraic models of programs provide a possibility to capture this effect. Let $Z : \mathcal{A} \to 2^{\mathcal{P}}$ be a mapping (invariant distribution) which associate with every basic instruction a a set of those basic predicates that are invariant w.r.t. a. We say that an evaluation function ξ *preserves* an invariant distribution Z if $(p \in \xi(s) \iff p \in \xi(s \circ a))$ holds for every data state s, $s \in S_0$, basic instruction a, $a \in \mathcal{A}$ and basic predicate $p \in Z(a)$. Denote by L_Z the set of all evaluation functions preserving an invariant distribution Z on the free semigroup S_0. As it was demonstrated in [16], ECP for any model $M_{0,Z} = (S_0, L_Z)$ has the same complexity as ECP for the maximal model.

Another effect which can be taken into account when refining the maximal model is that of monotonicity of predicates w.r.t. program instructions. For example, if the predicate `x>0` is true at some stage of program computation then it remains true after the execution of the instruction `x++`. We say that an evaluation function ξ is *monotonic* for a predicate p if $(p \in \xi(s) \implies p \in \xi(s \circ a))$ holds for every data state s, $s \in S_0$, and basic instruction a, $a \in \mathcal{A}$. Just as it was done above we may consider a family of algebraic models $\mathcal{M}_{0m} = (S_0, L_m)$ whose evaluation functions are monotonic for some basic predicates. In [36] it was shown that ECP for any monotonic refinement of maximal algebraic model is decidable in time $O(n \log n)$. Some other refinements of $\mathcal{M}_0$ were also studied in [36].

It is worthy of noticing that both properties considered above are well suited for the designing of *opaque constructs* [7, 8]. Opaque constructs are intended for hindering program understanding; they are widely used in program obfuscation for inserting dead code, modifying control and data flows, etc. Opaque constructs may be also employed by a metamorphic virus for "weaving" its code with a code of infected executable and for imitation of legal executables as well. These capabilities of obfuscating transformations have to be taken into account when designing advanced virus scanners.

3.2. Commutative semigroups. Models $\mathcal{M}_1 = (S_1, L_{S_1})$ of PSPs with commutative instructions.

The semantics of some program instructions is such that the result of their execution does not depend on the order in which these instructions are executed. To certify that instructions a and b can commute it is sufficient to check with the help of simple syntactic analyzer that $used(a) \cap mod(b) = used(b) \cap mod(a) = mod(a) \cap mod(b) = \emptyset$. This holds, for example, for the instructions `x++` and `z++=&y`. To account this effect one has to deal with (partially) commutative semigroups S_1 specified by a set of defining relationships $a \circ b = b \circ a$. An algebraic model of programs $\mathcal{M}_1 = (S_1, L_{S_1})$ based on a (partially) commutative semigroups S_1 is called a *commutative model*.

Code transposition based on the commutativity of instructions is one of the most simple and effective obfuscating transformation which can "put of the scent" all current pattern-matching virus scanners [4]. It shuffles the instructions so that their order in virus replicas is different from the order of instructions assumed in the signature used by the anti-virus software. Nevertheless an approach to the designing of efficient equivalence-checking algorithms for PSPs developed in [29, 38, 39] is powerful enough to cope with this problem.

Theorem 2. *ECP "$\pi_1 \sim_{\mathcal{M}_1} \pi_2$?" in the model $\mathcal{M}_1$ of PSPs with commutative instructions is decidable within a time $O(n^3 \log n)$, where n is the total size of PSPs π_1 and π_2.*

3.3. Models $\mathcal{M}_2 = (S_2, L_{S_2})$ of PSPs with suppressed instructions.

We say that a basic instruction a *suppresses* an instruction b if a result of execution of b is always discarded every time when a is executed next. Thus, for example, the instruction `x=y++` suppresses the instruction `x++`. To certify that an instruction a suppresses an instruction b one need only to check that $mod(b) \subseteq mod(a)$ and $mod(b) \cap used(a) = \emptyset$. The insertion/deletion of suppressed instructions is another kind of obfuscating transformation which enables a virus to stealthy change the length of its code (signature). Since the inserted instructions execute meaningful operations, this may mislead even those anti-virus tools that use a sandbox emulation approach.

The effect of instruction suppression is well represented by the family of algebraic models $\mathcal{M}_2 = (S_2, L_{S_2})$ whose semigroup S_2 is completely defined by the set of equalities of the form $b \circ a = a$. By applying the techniques developed in [38] we

proved the following

Theorem 3. *ECP "$\pi_1 \sim_{\mathcal{M}_2} \pi_2$?" in the model $\mathcal{M}_2$ of PSPs with suppressed instructions is decidable within a time $O(n^2 \log n)$, where n is the total size of PSPs π_1 and π_2.*

3.4. Semigroups with right zeros. Models $\mathcal{M}_3 = (S_3, L_{S_3})$ of PSPs with mode switching instructions.

A run of a program with *mode switching* is divided into two stages. In the first stage a program selects an appropriate mode of computation. Several modes may be tried in turn before making the ultimate choice. Every time when the next mode is put to the test, the program brings the data to some predefined state. In the second stage, once a definitive mode is fixed, the final result of computation is generated. In real programs mode switching may be implemented by restart instructions or constant assignments like `x=(y=0)+(z=1)`. The characteristic property of a mode switching instruction a is that $used(a) = \emptyset$, and $mod(a)$ includes all variables of the program. Since any execution of a mode switching instruction always gives the same result, the further behaviour of a program is well predictable. Therefore the mode switching instructions are used as a suitable means for generating opaque predicates whose outcome is known at obfuscation time, but is difficult for the obfuscator to deduce [7].

The effect of mode switching instructions is captured by the algebraic models of programs $\mathcal{M}_3 = (S_3, L_{S_3})$, where S_3 is a free semigroup with several right zeros, i.e. the elements e that satisfy the equation $a \circ e = e$ for every basic instruction a. In [19] it was proved that ECP for programs with mode switching instructions is decidable. A more efficient equivalence-checking algorithm for $\mathcal{M}_3$ was presented in [30].

Theorem 4. *ECP "$\pi_1 \sim_{\mathcal{M}_3} \pi_2$?" in the model $\mathcal{M}_3$ of PSPs with mode switching instructions is PSPACE-complete.*

3.5. Free groups. Models $\mathcal{M}_4 = (S_4, L_{S_4})$ of PSPs with invertible instructions.

Two instructions a and b are called (mutually) *invertible* if their successive execution always brings data to an original state. `x++` and `x-` is the most simple example of a pair of mutually invertible instructions. Invertible instructions, as well as constant assignments (or mode switching instructions), may be used for bringing data into some predefined states and, thus, for generating opaque constructs. But since

a data state which is reached after the execution of a sequence of mutually invertible instructions is not fixed (unlike the case of mode switching instructions), a superficial static analysis can't cope with this effect; it requires a more deep consideration that involves semantical features of program semantics.

The behaviour of programs with invertible instructions can be analyzed with the help of algebraic models of programs $\mathcal{M}_4 = (S_4, L_{S_4})$, where $S4$ is a free group. In [20, 21] it was shown that ECP for algebraic models of programs with invertible instructions is decidable. Recently we have estimated the complexity of ECP for $\mathcal{M}_4$.

Theorem 5. *ECP "$\pi_1 \sim_{\mathcal{M}_4} \pi_2$?" in the model $\mathcal{M}_4$ of PSPs with invertible instructions is PSPACE-complete.*

4. Equivalence-checking problem for compound algebraic models of programs

Usually to analyze program behaviour one has to take into account a number of various semantical properties of program instructions and predicates at once. This constrain us to consider more complex algebraic models of programs. Some of them stem from the combination of several basic models discussed above. It turns out, however, that joining together some rather simple algebraic models of programs we often obtain a compound model which is far more difficult for the analysis. Below we give some examples to illustrate this observation.

4.1. Algebraic models of PSPs with commutative instructions and monotonic predicates.

As it was shown in Subsections 3.1 ECP for algebraic models of programs with monotonic predicates is decidable in time $O(n \log n)$. In Subsection 3.2 we demonstrated also that the same problem for algebraic models with commutative instructions is decidable in time $O(n^3 \log n)$. In both cases the complexity of ECP does not depend significantly on the cardinality of the sets $\mathcal{A}$ and $\mathcal{P}$ of basic instructions and predicates. But if we consider a compound model $\mathcal{M}_{0m} + \mathcal{M}_1$ which combines commutative instructions and monotonic predicates then the best known equivalence-checking algorithm [41] for this model is not as much efficient as those for $\mathcal{M}_{0m}$ and $\mathcal{M}_1$.

Theorem 6. *ECP "$\pi_1 \sim_{\mathcal{M}_0} \pi_2$?" in the compound model $\mathcal{M}_{0m} + \mathcal{M}_1$ is decidable within a time $n^{O(rk)}$, where n is the total size of PSPs π_1 and π_2, and r and k are the cardinalities of the alphabets $\mathcal{A}$ and $\mathcal{P}$.*

ECP is even more complicated for algebraic models whose evaluation functions are subjected to invariant distributions Z. Let $M_{0Z} + M_1 = (S_1, L_{(S_1,Z)})$ be an algebraic model, where S_1 is a free commutative semigroup and Z is an invariant distribution such that $p \notin Z(a) \implies p \in Z(b)$ holds for every pair of basic instructions a, b, $a \neq b$, and predicate p. Then ECP for $M_{0Z} + M_1$ is inter-reducible with ECP for deterministic multi-tape automata. For more than 30 years it was not known whether the latter problem is decidable until in 1991 Harju and Karhumaki [14] presented an exponential time decision procedure which solves this problem. Nevertheless, to the extent of our knowledge, the complexity of ECP for deterministic multi-tape automata is still unclear.

4.2. Algebraic models of PSPs with commutative and mode switching instructions.

By joining together the algebraic models of programs introduced in Subsection 3.2 and Subsection 3.4 we obtain an algebraic model $\mathcal{M}_1 + \mathcal{M}_3$ whose Kripke structures are based on a free commutative semigroup augmented with several right zeros. ECP for this model was studied by Godlevsky; in [12] he presented an equivalence-checking algorithm for $\mathcal{M}_1 + \mathcal{M}_3$ which has super-exponential time complexity. The development of more efficient equivalence-checking procedures and more precise estimation of ECP complexity for this model is a topic of our future research.

One may consider also another of combination of $\mathcal{M}_1$ and $\mathcal{M}_3$ which leads to an algebraic model whose Kripke structures are based on a direct product of free commutative semigroups each supplied with right zeros. This is the case when program contains the instructions `x++`, `y++`, `x=0`, and `y=0`. In [13, 26] it was shown that ECP for such algebraic models of programs is undecidable.

4.3. Algebraic models of PSPs with commutative and invertible instructions.

If we combine the algebraic models introduced in Subsection 3.2 and Subsection 3.5 then we obtain a model whose Kripke structures are based on Abelian groups. This model is appropriate for programs that contains such pairs of invertible instructions as `x++`, `x-`, `y++`, `y-`, etc. each affecting its own variable. In [20] it was shown that ECP for algebraic models $\mathcal{M}_1 + \mathcal{M}_4$ is undecidable.

4.4. Algebraic models of PSPs with suppressed and commutative instructions.

Nevertheless, in some cases the complexity of ECP for a compound model does not exceeds considerably the complexity of ECP for its components. Thus, for example, we may consider an algebraic model $\mathcal{M}_1 + \mathcal{M}_3$ with suppressed and commutative instructions which is obtained by joining together the models introduced in Subsection 3.2 and Subsection 3.3.

Theorem 7. *ECP "$\pi_1 \sim_{\mathcal{M}_2} \pi_2$?" in the compound model $\mathcal{M}_1 + \mathcal{M}_3$ of PSPs with suppressed and commutative instructions is decidable within a time $O(n^4 \log n)$, where n is the total size of PSPs π_1 and π_2.*

5. Conclusion

As it can be seen from Theorems 1–3 and 7 in some cases equivalence-checking algorithms for algebraic models of programs are very efficient (have polynomial time complexity, and the power of polynomials is low). This means that the obfuscating transformations that are "visible" in such models are far from being secure. If a metamorphic virus employs only such transformations for obfuscation then its detection can be achieved through an advanced pattern-matching approach based on the uniform and efficient equivalence-checking procedures.

At the same time ECP for some algebraic models may be very hard and even intractable (see Theorems 4-6). Metamorphic viruses employing obfuscating transformations that are "visible" in these models may be very much stealthy. In these cases uniform equivalence-checking techniques give no effect. Fortunately, this also implies that the implementation of such obfuscating transformations is also a hard task which requires a sophisticated software analysis and consumes a considerable amount of computational resources. Since a computer virus, as a rule, has a compact code, it is hardly possible that it would employ very complex obfuscation.

Therefore, we assume that the following obfuscating strategy would be the most appropriate for metamorphic viruses. Suppose that $\mathcal{M} = \mathcal{M}_{i_1} + \mathcal{M}_{i_2} + \cdots + \mathcal{M}_{i_k}$ is a complex algebraic model which is composed of a number of simple models $\mathcal{M}_{i_1}, \mathcal{M}_{i_2}, \ldots, \mathcal{M}_{i_k}$. Suppose also that every model $\mathcal{M}_{i_j}$ admits the designing of compact and efficient engine for generating a sufficient amount of obfuscating transformations. Then to modify its code a virus π may apply randomly a series of obfuscating transformations $\mathcal{O}_1, \mathcal{O}_2, \ldots, \mathcal{O}_N$ that are "visible" in the framework of the models $\mathcal{M}_{i_1}, \mathcal{M}_{i_2}, \ldots, \mathcal{M}_{i_k}$. The order in which these transformations are applied is a "secrete key" of the obfuscation $\mathcal{O}$ thus obtained. If ECP for $\mathcal{M}$ is in-

tractable then a direct equivalence-checking approach to the deobfuscation of $\mathcal{O}(\pi)$ may happen to be time-consuming. One may avoid this difficulty by taking into account the arrangement of $\mathcal{M}$ and using deobfuscating procedures for each simple model $\mathcal{M}$ is composed of. But without knowing the "secrete key" of obfuscation it is also uneasy to find the proper order these procedures have to be applied to $\mathcal{O}(\pi)$. These considerations conform with the result obtained in the paper [33]

We think that further research on this topic would be very much useful for understanding the potency of metamorphic viruses and for designing advanced anti-virus software.

References

[1] B, Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vedhan, K. Yang. On the (Im)possibility of obfuscating programs. *CRYPTO'01 - Advances in Cryptology*, Lecture Notes in Computer Science, v. 2139, 2001, p. 1–18. 79

[2] D. Chess, S. White. An undetectable computer virus. In *Proc. of the 2000 Virus Bulletin Conference*, 2000. 79

[3] S. Chow, Y. Gu Y, H. Johnson, V.A. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. *6-th Inf. Security Conf.* Lecture Notes in Computer Science, V. 2200, 2001, p.144–155. 79

[4] M. Christodorescu, S. Jha. Static analysis of executables to detect malicious patterns. In *Proc. of the 12th USENIX Security Symposium (Security'03)*, 2003, p. 169–186. 80, 84, 86

[5] M. Christodorescu, S. Jha. Testing malware detectors. In *Proc. of the Int. Symp. on Software Testing and Analysis (ISSTA 2004)*, 2004. 79, 80

[6] M. Christodorescu, S. Jha, S.A. Seshia, D. Song, R. E. Bryant. Semantics-aware malware detection. In *Proc. of the 2005 IEEE Symp. on Security and Privacy (Oakland 2005)*, 2005. 79, 80, 84

[7] C. Collberg, C. Thomborson, D. Low. A taxonomy of obfuscating transformations. Tech. Report, N 148, Univ. of Auckland, 1997. 78, 79, 85, 87

[8] C. Collberg, C. Thomborson, D. Low. Manufacturing cheap, resilient and stealthy opaque constructs. *Symp. on Principles of Programming Languages*, 1998, p.184–196. 78, 79, 85

[9] P. Cousot, R. Cousot. An abstract interpretation-based framework for software watermarking. *Conf. Record of the 31st ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Programming Languages*, 2004, p. 173–185. 81

[10] M. Della Preda, R. Giacobazzi. Semantic-based code obfuscation by abstract interpretation. Lecture Notes in Computer Science, v. 3580, 2005, p. 1325–1336. 81

[11] M. Della Preda, R. Giacobazzi, M. Madou, B. de Bosschere. Opaque predicate detection by means of abstract interpretations. *Proc. of 11-th Int. Conf. on Algebraic Methodology and Software Technology (AMAST06)*, 2006. 81

[12] A.B.Godlevsky. On some specific cases of halting problem and equivalence problem for automata. *Cybernetics*, 1973, N 4, p.90–97 (in Russian). 89

[13] A.B.Godlevsky. On some special case of the functional equivalence problem for discrete transformers. *Cybernetics*, 1974, N 3, p.32–36 (in Russian). 89

[14] T. Harju, J. Karhumaki. The equivalence of multi-tape finite automata. *Theoretical Computer Science*, v. 78, 1991, p.347–355. 89

[15] J.E. Hopcroft, R.M. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report TR 71-114, Cornell University, Computer Science Dep., 1971. 81, 84

[16] Iu. I. Ianov. On the equivalence and transformation of program schemes. *Comm. of the ACM*, v.1, N 10, 1958, p. 8–12. 80, 81, 84, 85

[17] K.S. Ivanov, V.A. Zakharov. Program obfuscation as obstruction of program static analysis. *Proc. of the Institute for system programming*, v. 6, 2004, p. 141–161. 79

[18] E. Kaspersky. Virus List Encyclopedia. Kaspersky lab, 2002. 79

[19] A.A.Letichevsky. On the equivalence of automata with final states on the free monoids having right zeros. *Reports of the Soviet Academy of Science*, v. 182, N 5, 1968 (in Russian). 87

[20] A.A.Letichevsky. On the equivalence of automata over semigroup. *Theoretic Cybernetics*, v. 6, 1970, 3–71 (in Russian). 80, 88, 89

[21] A.A.Letichevsky, L.B.Smikun. On a class of groups with solvable problem of automata equivalence. *Sov. Math. Dokl.*, 17, 1976, N 2, 341—344. 88

[22] A. Marx. A guideline to anti-malware-software testing. *Proc. of the 9th Ann. European Institute for Computer Antivirus Research Conference (EICAR'00)*, 2000. 78

[23] G. McGrow, G. Morriset. Attacking malicious code: report to the Infosec research council. *IEEE Software*, 2000, v. 17, N 5, p. 33–41. 77

[24] C. Nachenberg. Understanding and Managing Polymorphic Viruses. *The Symantec Enterprise Papers*, v. XXX, 1996, 16 p. 78

[25] C. Nachenberg. Computer Virus-Antivirus Coevolution. *Comm. of the ACM*, v. 40, N. 1, 1997, p.46-51. 78

[26] G.N.Petrosyan. On one basis of statements and predicates for which the emptiness problem is undecidable. *Cybernetics*, 1974, N 5, p. 23–28 (in Russian). 89

[27] R.I.Podlovchenko. The hierarchy of program models. *Programming and Computer Software*, 1981, N 2, p. 3-–14 (in Russian). 80, 83

[28] R.I.Podlovchenko. Semigroup program models. *Programming and Computer Software*, 1981, N 4, p. 3—13 (in Russian). 80

[29] R.I.Podlovchenko, V.A.Zakharov. On the polynomial-time algorithm deciding the commutative equivalence of program schemata. *Reports of the Russian Academy of Science*, **362**, 1998, N 6 (in Russian). 81, 86

[30] R. Podlovchenko, D. Rusakov D, V. Zakharov. On the equivalence problem for programs with mode switching. *Conf. on the Implementation and Application of Automata*, Lecture Notes in Computer Science, v. 3845, 2006, p. 351–352. 87

[31] J.D.Rutledge. On Ianov's program schemata. *Journal of the ACM*, v. 11, 1964, p.1–9. 84

[32] T. Sampson. A virus in info-space: the open network and its enemies. *Journal of Media and Culture*, 2004, v. 7, N 3. 77

[33] D. Spinellis. Reliable identification of bounded-length viruses is NP-complete. *IEEE Trans. on Information Theory*, v. 49, N 1, 2003, p. 280-–284. 91

[34] P. Szor, P. Ferrie. Hunting for metamorphic. In *Proc. of the 2001 Virus Bulletin Conference*, 2001, p. 123–144. 79

[35] N.P. Varnovsky, V.A. Zakharov. On the possibility of provably secure obfuscating programs. *5-th Conf. "Perspectives of System Informatics"*, Lecture Notes in Computer Science, v. 2890, 2004, p. 91–102. 79

[36] V.A. Zakharov. On automata program models. *Reports of the USSR Academy of Sciences*, 1989, v. 309, N 1, p. 24–27 (in Russian). 85

[37] V.A. Zakharov. On the comparability criteria for formal models of imperative programs. *Programming and Computer Software*, 1993, N 4, p. 12—25 (in Russian). 80, 83

[38] V.A. Zakharov. An efficient and unified approach to the decidability of equivalence of propositional program schemes. Lecture Notes in Computer Science, v. 1443, 1998, p. 247–258. 81, 86

[39] V.A. Zakharov. On the decidability of the equivalence problem for monadic recursive programs. *Theoretical Informatics and Applications*, v. 34, N 2, 2000, p. 157-–171. 86

[40] V.A. Zakharov. The equivalence problem for computational models: decidable and undecidable cases. Lecture Notes in Computer Science, v. 2055, 2001, p. 133–153. 80, 81

[41] V.A. Zakharov, I.M. Zakharyaschev. On the equivalence checking problem for a model of programs related with multi-tape automata. *Conf. on Implementation and Application of Automata*, Lecture Notes in Computer Science, v. 3317, 2005, p. 293-–305. 88