Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 331 (2017) 87–99

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Abstract Similarity Analysis

Mila Dalla Preda and Vanessa Vidali

Dipartimento di Informatica, Università degli studi di Verona, Italy

Abstract

Code similarity is an important component of program analysis that finds application in many fields of computer science. Graph based representations of programs, such as control flow graphs and dependency graphs, are often used as a basis for deciding code similarity. Indeed, many similarity algorithms observe particular properties of these graph-based representations of programs in order to decide whether two programs are similar or not. In this work we propose a general framework for similarity analysis where the similarity of programs is expressed in terms of abstractions of their control flow graphs representation. In particular, we consider abstractions of the basic blocks of a control flow graph.

Keywords: Code similarity, abstract interpretation.

1 Introduction

Code similarity studies if two programs are similar or if one program is similar to a portion of another program (code containment). Code similarity is an important component of program analysis that finds application in many fields of computer science, such as reverse engineering of big collections of code fragments [13,16], clone detection [2,8], identification of violations of the intellectual property of programs [1,17,12], malware detection [9,10,15], software maintenance [11,19], software forensics [2,14]. In these applications, when comparing two fragments of code it is important to take into account changes due to code evolution, compiler optimization and post-compile obfuscation. These code changes give rise to fragments of code that are syntactically different while having the same intended behavior. This means that it is important to recognize modifications of the same program that are obtained though compiler optimization or code obfuscation as similar. To this end we need to abstract from syntactic changes and implementation details that do not alter the intended behavior of programs, namely that preserve to some extent the semantics of programs.

[1] Email: mila.dallapreda@univr.it, vanessa.vidali@studenti.univr.it

[2] This work has been supported by the MIUR FIRB 2013 project FACE RBFR13AJFT.

http://dx.doi.org/10.1016/j.entcs.2017.02.006

In order to consider both semantic meanings and syntactic patterns, existing tools for similarity analysis often employ mixed syntactic/symbolic and semantic representations of programs, as for example control flow graphs and dependency graphs that express the flow of control or the dependencies among program instructions. Recently, in [7] the authors investigate the use of symbolic finite automata (SFA) and their abstractions for the analysis of code similarity. SFAs have been introduced in [18] as an extension of traditional finite state automata for modeling languages with a potential infinite alphabet. Transitions in a SFA are modeled as constraints interpreted in a given Boolean algebra, providing the semantic interpretation of constraints, and therefore the (potentially infinite) structural components of the language recognized (see [5,18]). In [7] the authors show how SFAs can be used to represent both the syntax and the semantics of programs written in an arbitrary programming language, the idea is to label transitions with syntactic labels representing program instructions, while their interpretation is given by the semantics of such instructions. Thus, SFAs provide the ideal formal setting in order to treat within the same model the abstraction of both the syntactic structure of programs and their intended semantics. A formal framework for the abstraction of syntactic and semantic properties of SFAs and therefore of programs represented as SFAs is presented in [7]. This formal framework turns out to be very useful in the understanding of existing similarity analysis tools, and in the development of similarity analysis tools based on semantic and syntactic properties of programs.

The work presented in this paper and the prototype implementation that we propose can be seen as a first step towards an experimental validation of the general theory for software similarity proposed in [7]. In this paper we consider the standard control flow graph (CFG) representation of programs that is a simplified model with respect to the SFA representation of programs proposed in [7]. Indeed, a CFG can be easily translated into an SFA where the automata is isomorphic to the CFG, basic blocks label the automata transitions and the interpretation of the basic blocks is the identity function (this is because the CFG representation of programs does not account for the interpretation of basic blocks). In the attempt to build a similarity analysis tool parametric on program's properties we decide to start form the simpler case of CFGs. Thus, in this work we present a general framework for code similarity where the notion of being similar is formalized in terms of abstraction, in the abstract interpretation sense, of the CFG of programs. We propose a methodology for testing code similarity that is parametric on the property of the basic blocks of the CFG that is used for deciding similarity. Indeed, many existing algorithms for similarity analysis can be interpreted in our framework by formalizing the abstract property of the blocks of the CFG that they consider. This allows us to compare existing similarity algorithms by comparing the abstractions that characterize them. Moreover, the precision of the proposed similarity test can be tuned by modifying the abstract property of the CFG that is being observed. We provide a prototype implementation of the proposed methodology that allows us to test the similarity of two fragments of code with respect to three different abstract properties of basic blocks.

Structure of the paper: Section 2 recalls the basic notions used in the paper. In Section 3 we present a similarity analysis framework based on the abstract interpretation of the CFG. In Section 4 we present the details of our implementation and discuss the results that we have obtained. The paper ends with a discussion on future work.

2 Background

Mathematical notation. Given a set S, we denote with $\wp(S)$ the powerset of S. $\langle P, \leq\rangle$ denotes a poset P with ordering relation $\leq$, while a complete lattice P, with ordering relation $\leq$, least upper bound (lub) $\vee$, greatest lower bound (glb) $\wedge$, greatest element $\top$, and least element $\bot$ is denoted by $\langle P, \leq, \vee, \wedge, \top, \bot\rangle$. $\sqsubseteq$ denotes the pointwise ordering between functions. $f : C \to D$ on complete lattices is additive (resp. co-additive) when, for any $Y \subseteq C, f(\vee Y) = \vee f(Y)$ ($f(\wedge Y) = \wedge f(Y)$). The right (resp. left) adjoint of a function f is $f^{+} \stackrel{\text{def}}{=} \lambda x. \bigvee\{y \mid f(y) \leq x\}$ [$f^{-} \stackrel{\text{def}}{=} \lambda x. \bigwedge\{y \mid x \leq f(y)\}$].

A directed graph is defined by an ordered pair $G = (N, E)$ where N is the set of nodes and $E \subseteq N \times N$ is a set of ordered pairs that represent the edges of the graph. A subgraph of a directed graph $G = (N, E)$ is a directed graph $G' = (N', E')$ where $N' \subseteq N$ and $E' \subseteq E$. We denote with $Sub(G)$ the set of possible subgraphs of graph G. Two graphs $G_1 = (N_1, E_1)$ and $G_2 = (N_2, E_2)$ are isomorphic if there exists a bjection $f : N_1 \to N_2$ between their nodes and $(n_1, n_1') \in E_1 \Leftrightarrow (n_2, n_2') \in E_2$, where $f(n_1) = n_2$ and $f(n_1') = n_2'$. We denote with $|G|$ the number of nodes of graph G.

Abstract Interpretation. Abstract interpretation is used for reasoning on properties rather than on (concrete) data values [3]. Approximation can be equivalently formulated either in terms of Galois connections or closure operators [4]. A Galois connection (GC) is a tuple (C, α, γ, A), where $\langle C, \leq_C\rangle$ is the concrete domain, $\langle A, \leq_A\rangle$ the abstract domain, $\alpha : C \to A$ the abstraction map and $\gamma : A \to C$ the concretization map that form an adjunction: $\forall y \in A, x \in C : \alpha(x) \leq_A y \Leftrightarrow x \leq_C \gamma(y)$ [3,4]. α (resp. γ) is the left(resp. right)-adjoint of γ (resp. α) and it is additive (resp. co-additive, i.e. it preserves the lub (resp. glb) of all the subsets of the domain (empty set included). Recall that a tuple (C, α, γ, A) is a GC iff α is additive iff γ is co-additive. Indeed, an additive abstraction map α induces a concretization map γ and vice versa, formally $\gamma(y) = \vee\{x \mid \alpha(x_{\leq} y\}$ and $\alpha(x) = \wedge\{y \mid x \leq \gamma(y)\}$. An upper closure operator, or closure, on poset $\langle C, \leq\rangle$ is an operator $\varphi : C \to C$ that is monotone, idempotent and extensive (i.e. $\forall c \in C : c \leq \varphi(c)$). Closures are uniquely determined by the set of their fixpoints $\varphi(C)$. The set of all closure on C is denoted by $uco(C)$. The lattice of abstract domains of C is therefore isomorphic to $uco(C)$ [3,4]. If C is a complete lattice, then $\langle uco(C), \sqsubseteq, \sqcup, \sqcap, \lambda x.\top, id\rangle$ is a complete lattice, where $id \stackrel{\text{def}}{=} \lambda x.x$ and for every $\rho, \eta \in uco(C)$, $\rho \sqsubseteq \eta$ iff $\forall y \in C : \rho(y) \leq \eta(y)$ iff $\eta(C) \subseteq \rho(C)$. If (C, α, γ, A) is a GC then $\varphi = \gamma \circ \alpha$ is the closure associated with A, such that $\varphi(C)$ is a complete lattice isomorphic to A.

3 Abstract Similarity Analysis Framework

We consider programs represented as CFGs. The CFG of a program is a graph where nodes are given by sequences of non branching instructions and edges represent possible control flows. More formally, let $\mathbb{I}$ be the set of possible program instructions ranged over by $\mathtt{i}$, we define the set $\mathbb{B} \stackrel{\text{def}}{=} \{\mathtt{b} = \mathtt{i}_1 \dots \mathtt{i}_n \mid \forall i \in [1,n] : \mathtt{i}_i \in \mathbb{I}, \mathtt{i}_i \text{ is a non benching instruction } \}$, representing possible basic blocks. The CFG of a program $P \in \mathbb{I}^*$ is a graph $G_P = (N_P, E_P)$ where the set $N_P \subseteq \mathbb{B}$ of nodes specifies the basic blocks of P and the edges $E_P \subseteq N_P \times N_P$ denote the possible flows of control during the execution of program P.

Consider the concrete domain of the powerset of basic blocks $\langle \wp(\mathbb{B}), \subseteq \rangle$ and an abstract domain $\langle \hat{\mathbb{B}}, \hat{\leq} \rangle$ related by an additive abstract function $\alpha : \wp(\mathbb{B}) \to \hat{\mathbb{B}}$ that gives rise to a Galois connection $\langle \wp(\mathbb{B}), \alpha, \gamma, \hat{\mathbb{B}} \rangle$. Given an abstraction function α on basic blocks we say that two CFGs G and G' are α-*equivalent*, denoted $G \equiv_\alpha G'$, if they are isomorphic graphs and the corresponding basic blocks are abstracted by α to the same abstract basic blocks. Given an abstraction of basic blocks it is possible to derive an abstraction of CFGs expressed as a closure operator on the powerset domain of CFGs ordered with respect to set inclusion, i.e., $\langle \wp(\mathit{CFG}), \subseteq \rangle$.

Definition 3.1 A block abstraction $\alpha : \wp(\mathbb{B}) \to \hat{\mathbb{B}}$ induces a closure $\rho_\alpha \in \mathit{uco}(\wp(\mathit{CFG}))$: $\rho_\alpha \stackrel{\text{def}}{=} \lambda X.\{G' \in \mathit{CFG} \mid \exists G \in X : G \equiv_\alpha G'\}$.

Given a basic block abstraction $\alpha : \wp(\mathbb{B}) \to \hat{\mathbb{B}}$, the CFG abstraction defined by ρ_α groups together the CFGs that are α-equivalent, namely that have the same shape and the corresponding basic blocks are indistinguishable with respect to α. Observe that two CFGs that are α-equivalent correspond to two programs that have the same control structure and corresponding basic blocks that are abstracted to the same abstract object by block abstraction α. We use the notion of α-equivalence to define the following tests for CFG similarity.

Definition 3.2 Consider $\alpha : \wp(\mathbb{B}) \to \hat{\mathbb{B}}$ and two CFGs G_1 and G_2 we say that:

- G_1 and G_2 are α-*similar* iff $\rho_\alpha(G_1) = \rho_\alpha(G_2)$, namely if $G_1 \equiv_\alpha G_2$,
- G_1 is α-*contained* in G_2, denoted $G_1 \preceq_\alpha G_2$ iff there exists $H_2 \in \mathit{Sub}(G_2)$ such that $\rho_\alpha(G_1) = \rho_\alpha(H_2)$, namely if $G_1 \equiv_\alpha H_2$.

It is clear that α-similarity classifies as similar CFGs that are α-equivalent, while α-containment verifies α-similarity between a CFG and the possible sub-graphs of the other CFG. The parameter α fixes the properties of basic blocks to observe in order to decide similarity among isomorphic CFGs. Of course, if α considers syntactic properties of the basic blocks the similarity tests will be sensitive to syntactic changes, while an abstraction α that abstracts from syntactic details and looks at semantic properties of the basic blocks will be more resilient to syntactic code variations (induced for example by code obfuscation). However, this flexibility is only block-wise. Indeed, the α-similarity test is very rigid with respect to the control flow structure of programs, while it is flexible with respect to the fragments of code in the basic blocks. This allows us to recognize as similar only

programs that are obtained through syntactic transformations, i.e., code obfuscations and compiler optimizations, that work block-wise. For example, α-similarity does not recognize as similar programs that are obtained through semantic preserving control flow transformations like opaque predicate insertion, loop unrolling, code motion for loop optimization and so on. On the other hand, by choosing a suitable block abstraction α, the α-similarity test allows us to recognize as similar programs that are obtained through code transformations that work block-wise, like variable renaming, expression reshaping and so on.

It is clear that the precision of the abstract similarity test depends on the choice of the abstraction α, and therefore on the closure ρ_α that it induces. If $\alpha = \lambda X.\top$ then the proposed methodology classifies as similar all the CFGs that have the same shape regardless of the code present in the basic blocks. On the other hand if $\alpha = \lambda X.X$ then it means that the proposed methodology classifies as similar only CFGs that have the same shape and exactly the same basic blocks. In general, if $\alpha_1 \sqsubseteq \alpha_2$ it means that when two CFGs are α_1-similar then they are also α_2-similar but not vice versa. This means that in the proposed formal framework for abstract similarity analysis it is possible to compare different similarity tools, with respect to the relative degree of precision of the abstractions of basic block used. Existing tools for similarity analysis that work on CFGs can be compared in the proposed framework by specifying the abstraction of CFGs that they implicitly use.

Given a program transformation $T : \mathit{Prog} \to \mathit{Prog}$ and a basic block abstraction $\alpha : \wp(\mathbb{B}) \to \hat{\mathbb{B}}$, we say that the test of α-similarity is *insensitive* to transformation T, if $\forall P \in \mathit{Prog}$ we have that $G_P \equiv_\alpha G_{T(P)}$, meaning that the α-similarity test recognizes that P and $T(P)$ are mutations of the same program. Interestingly, in the proposed formal framework for similarity analysis it is possible to formally characterize the class of program transformations for which a certain α-similarity test is insensitive. We denote with $\delta_T \in \mathit{uco}(\wp(\mathit{CFG}))$ the most concrete property preserved by transformation $T : \mathit{Prog} \to \mathit{Prog}$ on the CFGs of programs, namely: $\delta_T \stackrel{\text{def}}{=} \sqcap \{\delta \in \mathit{uco}(\wp(\mathit{CFG})) \mid \forall P \in \mathit{Prog} : \delta(G_P) = \delta(G_{T(P)})\}$. Recall that in [6] the authors provide a systematic methodolgy for deriving the most concrete property δ_T preserved by a program transformation T. Thus, given a basic block abstraction $\alpha : \wp(\mathbb{B}) \to \hat{\mathbb{B}}$ we define the set of program transformations for which α-similarity is insensitive as:

$$\mathcal{T}_\alpha \stackrel{\text{def}}{=} \{T : \mathit{Prog} \to \mathit{Prog} \mid \rho_\alpha \sqsubseteq \delta_T\}$$

This means that the modifications of a program P obtained through transformations in $\mathcal{T}_\alpha$ are α-similar, as stated in the following:

Theorem 3.3 *Consider $\alpha : \wp(\mathbb{B}) \to \hat{\mathbb{B}}$ and $T \in \mathcal{T}_\alpha$, then for any program P we have that G_P and $G_{T(P)}$ are α-similar, namely $G_P \equiv_\alpha G_{T(P)}$.*

Figure 1 represents the workflow for the analysis of code similarity based on the three following modules: (1) a *CFG extractor*, (2) an *abstraction engine* and (3) a *similarity check*. Assume to have a set of programs that we want to classify with respect to their similarity. The first thing that we have to do is to extract the CFGs of these programs by using either a static or dynamic approach. Next, we need

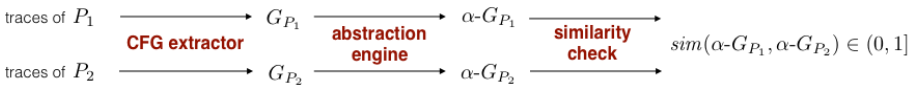

Fig. 1. Workflow and modules of an abstract similarity analysis tool

to fix the abstract property α of basic blocks that we want to use and we have to automatically abstract the basic blocks of these CFGs. Finally the similarity check is done on the abstract CFGs and it can be used for grouping together CFGs that are α-similar.

4 Prototype Implementation

We have implemented the similarity analysis methodology described in Figure 1. In the following we provide a description of our implementation and some experiments.

CFG Extractor: We have implemented an algorithm that dynamically extracts CFGs of programs from corresponding execution traces, thus studying what is effectively executed and the effects on the system. This may be useful in the analysis of malicious programs where the code may be packed, or in the analysis of code that employs code obfuscation such as opaque predicate insertion for impeding reverse engineering. Indeed, by analyzing the execution trace of a program that contains an encrypted section it is possible to discover and comprehend the encryption/decryption engine and sometimes also the used keys.

We used program *tracing* to examine the execution trace off-line, after the program has finished running. Off-line investigation gives less control than interactive debugging but allows us to have a view on the whole trace. Indeed, by program tracing we were able to collect not only how many times each basic block or control flow edge has been traversed like in profiling, but also the actual list of blocks or edges that have been executed. In particular, we consider *finite dynamic trace* as a record of the sequence of operations and instructions encountered during execution on a certain input. Formally, a finite dynamic trace can be expressed as a set $trace = \langle \mathtt{i}_i \ldots, \mathtt{i}_j \rangle$, where $\mathtt{i}_i \ldots \mathtt{i}_j$ are instructions in P. A *tracer* is a program *Tracer* that takes in input a program P and a set of its inputs X and, executing P over X, returns the set of traces representing the sequences of executed instructions. Thus, given a program P and inputs $x_1, \ldots, x_n \in X$: $Tracer(P, x_1) = trace_1, \ldots, Tracer(P, x_n) = trace_n$.

In our implementation, we collect finite dynamic traces using a timer to end the computation after a finite amount of time in case of non-termination. Each trace is then represented as a single graph, where each instruction is contained in a node that is identified by the line number of the instruction and the execution flow is given by directed edges. If the trace passes from the same point more than once, no node is added but the existing ones are used and reconnected, because of unique node identifications. This allows us to represent loops. These graphs representing the single dynamic traces are then merged together to create a unique graph, through node identifications. At the end, the graph is commuted into a CFG by

grouping nodes into basic blocks, where every basic block contains the sequence of instructions with no jumps. The proposed CFG extractor is based on dynamic analysis and for this reason it may not cover all possible program executions.

Abstraction engine: Once the CFG has been extracted, the abstraction can be performed. In our implementation we have considered basic block abstractions that work on single instructions. The main idea is to abstract the instructions in the basic blocks using typed symbolic variables in order to be independent from variable names. This is done by consistently replacing variable names with symbolic variables inside a block. The replacement is consistent in that two occurrences of the same variable in the same block are replaced by the same variable. In addition to abstracting the variable names, also constants are abstracted. This can be expressed as a block abstraction $\alpha : \wp(\mathbb{B}) \to \hat{\mathbb{B}}$, where each basic block is abstracted according to an instruction abstraction $\beta : \mathbb{I} \to \hat{\mathbb{I}}$ (where $\hat{\mathbb{I}}$ is a set of abstract instructions) as follows:

$$\alpha(\mathtt{ib}) = \beta(\mathtt{i}) \cup \alpha(\mathtt{b}) \qquad\qquad \alpha(\epsilon) = \emptyset$$

We will consider instruction abstractions $\beta : \mathbb{I} \to \hat{\mathbb{I}}$ that observe only assignment instructions, formally $\beta(\mathtt{i}) \in \hat{\mathbb{I}}$ if `i` is an assignment instruction, while $\beta(\mathtt{i}) = \emptyset$ otherwise. We have implemented three different instructions abstractions, these abstractions have a common base and every new version retains improvements from the previous one. Let us recall that an assignment instruction evaluates an expression, *right hand-side*, and assigns the resulting value to a target, *left hand-side*. We consider variables of type `string` and of type `number` (including `integers`, `floats` and `booleans`), while the other types are set to `unknown`. In our abstractions every constant is abstracted to its type: for example, `"hello world"` is abstracted to `str` (`string`) while `3` is abstracted to `numb` (`number`). Instead, every variable is abstracted into a symbolic name (in the following we use capital letters to denote symbolic names of variables). We assign a default value to the variables present in the *right hand-side* that are not previously declared or used. For example, if a basic block contains the assignment `y = x` but there are not previously uses of `x`, the abstraction rewrites this assignment to `B = def[A]`, where `A` and `B` are the symbolic names of variables and `def[A]` denotes the default value of the symbolic variable `A`. Moreover, we consider function composition as a single function: `fun1(fun2(x), y)` can be seen as `fun'(x,y)`. Let us now describe the instruction abstractions that we have implemented.

Instruction Abstraction β_1. This abstraction uses regular expressions to distinguish between expressions and functions in the right hand-side. We build two disjoint sets of bindings between names and symbolic objects: *varName* that collects the names of variables with the associated symbolic name and *funcName* that collects the function names with the associated symbolic names. The two sets are disjoint. When processing an assignment we start with the right hand-side. In this way we can recognize the default values and the possible expression operators. If it contains a number or a string, then this value is replaced with the tags `numb` or `str`. If there is an already used variable, its corresponding symbolic name is

restored from the *varName* set; otherwise we create a new binding between the variable and its new symbolic name that we add to *varName*. The same process is performed for functions, where function names are stored in *funcName*. The procedure is then repeated on the left hand-side. In the second column of Table 1 we report some examples of basic blocks that have been abstracted according to instruction abstraction β_1. For example, in the abstraction of the first basic block we can observe that the symbolic name `A` is bound to variable `x` on the left hand-side of the first assignment and on the right hand-side of the third assignment. Moreover, the abstraction recognizes that variable `a` in the right hand-side of the last assignment has not been defined before and therefore it has to refer to a default value.

Instruction Abstraction β_2. This abstraction is related to a specific feature of the programming language that we have considered. When programming in Python we observed that function names can be associated with a variable and then called using it. For example, we can rename a print function `print("Hello world")` in this way: first we create a variable `a = "print"` and then call `a("Hello world")`, and the result will be the same. So it becomes easy to obfuscate also the names of function. Therefore, we decided to modify the first abstraction by putting function names also in *varName*. This can be observed in the abstraction of the last basic block in Table 1. This basic block assigns to variable `x` and then uses variable `x` as an argument in the assignment `u = f(x)` and as a function in `v = x(3)`. We can observe that the basic block abstraction α_1 (induced by instruction abstraction β_1) does not recognize that in the last statement the function called is `x` and therefore uses a new symbolic name for function `fun0`. While the basic block abstraction α_2 (induced by instruction abstraction β_2) captures this dependence and in both cases abstracts `x` to the symbolic name `B`. On the other hand, this abstraction looses the distinction between variable names and function names.

Instruction Abstraction β_3. In this last attempt, we refine abstraction β_1 by keeping some information regarding the syntax of the expression of the right hand-side and by annotating symbolic names with type information. We do not explicitly mention the operators used in the expressions, but we use a tag `expr<args>` to indicate all numeric, string and function expressions, where `<args>` is the list of symbolic objects used in the expression. Since this could lead to a loss of information about the type of expressions, we decided to explicitly add type information. We assume a type priority: `string >> number >> unknown`, which means that, in one expression with strings, numbers and something else, the entire expression is considered as a `string` and so on. Therefore, the abstract syntax for assignment instructions becomes `var : type = expr< args >`. For example in the last column of Table 1 we can see how the last assignment `z = x + a` of the first basic block is abstracted in `D:number=expr<A,def[C]>`. We use an inference system to assign types to symbolic names. It is clear that even if the three proposed abstractions are related they also have their own peculiarities. This simple example shows the flexibility of our similarity test methodology with regards to the chosen basic block abstraction. We use the term abstract CFG to refer to the CFG where

Basic Block	Abstraction α_1	Abstraction α_2	Abstraction α_3
`x = 1`	`A = numb`	`A = numb`	`A:number=expr<>`
`y = 5+2*3`	`B = numb+numb*numb`	`B = numb+numb*numb`	`B:number=expr<>`
`z = x + a`	`D = A + def[C]`	`D = A + def[C]`	`D:number=expr<A,def[C]>`
`x = a`	`B = def[A]`	`B = def[A]`	`B:unknown=expr<def[A]>`
`a = 5`	`A = numb`	`A = numb`	`A:number=expr<>`
`x = "a"`	`A = str`	`A = str`	`A:string=expr<>`
`y = "a"+b"`	`B = str+str`	`B = str+str`	`B:string=expr<>`
`z = "a"+5`	`C = str+int`	`C = str+int`	`C:string=expr<>`
`v = a`	`D = A`	`D = A`	`D:string=expr<A>`
`w = s`	`F = def[E]`	`F = def[E]`	`F:string=expr<def[E]>`
`x = f(3)`	`A = fun0(numb)`	`B = A(numb)`	`B:unknow=expr<A()>`
`y = f(3)+2`	`B = fun0(numb)+numb`	`C = A(numb)+numb`	`C:number=expr<A()>`
`z = f(3)+"a"`	`C = fun0(numb)+str`	`D = A(numb)+str`	`D:string=expr<A()>`
`s = g(3, y)`	`D = fun1(numb, B)`	`F = E(numb, C)`	`F:unknown=expr<E(),C>`
`t = f(b)`	`F = fun0(def[E])`	`H = A(def[G])`	`H:unknown=expr<A(def[G]))>`
`u = f(x)`	`G = fun0(A)`	`I = A(B)`	`I:unknown=expr<A(B)>`
`v = x(3)`	`H = fun2(numb)`	`J = B(numb)`	`J:unknown=expr<B()>`

Table 1
Example of basic blocks abstractions α_i induced by the instruction abstraction β_i, with $i \in \{1, 2, 3\}$

each basic block has been abstracted according to a certain basic block abstraction.

Similarity check: When checking the similarity between two abstract CFGs we start by verifying whether the two abstract CFGs are isomorphic or not. If they are not isomorphic we check whether one is isomorphic to a subgraph of the other one. When a possible matching $M \subseteq N_1 \times N_2$ is found between two abstract CFGs $G_1 = (N_1, E_1)$ and $G_2 = (N_2, E_2)$, we continue by analyzing the *block similarity*. Block similarity is performed over single couples of matching nodes (m, n) where $m \in N_1$ and $n \in N_2$:

$$block\_similarity(m,n) = \begin{cases} 1 & m = \emptyset \wedge n = \emptyset \\ \dfrac{\mid m \cap n \mid}{\mid m \cup n \mid} & otherwise \end{cases} \tag{1}$$

Observe that, since the basic block abstractions that we have considered work on single instruction, we can measure the block similarity as the number of common abstract instructions. In this way we treat basic block as sets of instructions, thus loosing any information on the order in which these instructions are performed. Next, *global similarity* of the graphs is computed by summing up the block-similarities with respect to the maximal size of the considered graphs.

$$global\_similarity(M) = \frac{\sum_{(m,n)\in M} block\_similarity(m,n)}{max(|G_1|, |G_2|)} \tag{2}$$

Observe that $global\_similarity(M) \in [0, 1]$. Similarity can then be decided with

respect to an empiric *threshold* $\omega \in [0,1]$ calculated through experiments:

$$test\_similarity(G_1, G_2) = \begin{cases} YES & global\_similarity(M) \geq \omega \\ NO & otherwise \end{cases} \quad (3)$$

Implementation Details: The project has been developed in Python for the analysis of Python programs. This is because we were interested in dynamic extraction of CFGs and Python offers a simple module, called *Trace* [20], which allows us to trace program execution, generate annotated statement coverage listings, print caller/callee relationships and list functions executed during a program run without instrumenting the program. Another motivation that guided our choice towards Python concerns the need to study graph isomorphism. Indeed, we use NetworkX [21], a Python package for the creation, manipulation and study of structure, dynamics and functions of complex networks. In conjunction with Pydot, an interface to the DOT language for creating, handling, modifying and processing graphs, it provides also graph drawing and graph layout algorithms. We decided to use it also because it has two significant advantages over other graph analysis libraries: (1) it can read and write dot files directly (not native support but it translates them to its native graph object), and (2) it offers an approximation algorithm for graph (sub)isomorphism testing [22]: the implementation of VF2 Algorithm[23]. Briefly, given two graphs G and H, this procedure tries to match if G is subgraph for H and vice versa, and, if they are isomorphic, it returns the matching. Moreover, Python offers also a module for code obfuscation that we needed for testing the proposed similarity tool. Indeed, the obfuscation techniques that we have used in our tests are the ones offered by the Python module *PyMinifier* [24] such as the obfuscation of variable and function names and the reordering of instructions.

Experiments and Results: We have validated our prototype for the analysis of similarity on the following classes of programs:

- *Obfuscated programs:* in this case we compare programs and their obfuscated variants;
- *Infected programs:* in this case we developed a toy *virus* that replicates itself at the beginning of each infected Python program, and then we compare this virus with the infected programs and their obfuscations.

In particular, in our experiments we tested the similarity of pairs of programs using the three instruction abstractions proposed and checking for both α-similarity (i.e., isomorphism) and α-containment (i.e., sub-graph isomorphism).

In Table 2, we present the results of some of the experiments that we have run. The programs that we have considered are two simple programs `GCD1` and `GCD2` that compute the grater common divisor (GCD). In particular, `GCD1` is contained in `GCD2`, since `GCD2` contains more interactions with a possible user. We consider also two completely different versions of Fibonacci's algorithm: iterative `FibIterat` and recursive `FibRecurs`, and the toy virus `Virus` that we have developed. The prefix `obf-` in programs denotes obfuscated variants, while the prefix `inf-` relates to the

infected forms. So for example `inf-GCD1` represents program `GCD1` infected by the virus `Virus`, and `obf-GCD1` denotes program `GCD1` obfuscated with the obfuscations procedures provided by the Python module *PyMinifier* (i.e. renaming of variables and functions and instruction reordering). The first two columns show the two programs that we are testing for similarity, while the three other columns show for each abstraction the results of the similarity test obtained by our prototype implementation. For each basic block abstraction on the left side of the column we report the global similarity value when considering isomorphism over the entire graph, while on the right side of the column we report the similarity results for sub-graph isomorphism.

Program 1	Program 2	Abstr α_1		Abstr α_2		Abstr α_3	
		Isomorf	Sub-Isomorf	Isomorf	Sub-Isomorf	Isomorf	Sub-Isomorf
FibIterat	FibRecurs	0	0	0	0	0	0
GCD1	GCD2	0	0.4	0	0.4	0	0.47
Virus	inf-GCD1	0	0.76	0	0.76	0	0.78
Virus	inf-GCD2	0	0.75	0	0.75	0	0.75
GCD1	inf-GCD1	0	0.68	0	0.68	0	0.67
GCD2	inf-GCD2	0	0.66	0	0.66	0	0.66
inf-GCD1	inf-GCD2	0	0.3	0	0.3	0	0.33
GCD1	obf-GCD1	0.86	0.86	0.86	0.86	0.88	0.88
GCD2	obf-GCD2	0.82	0.82	0.83	0.83	0.83	0.83
Virus	obf-Virus	0.85	0.85	0.85	0.85	0.85	0.85
FibRecurs	obf-FibRecurs	0.84	0.84	0.84	0.84	0.84	0.84
FibIterat	obf-FibIterat	0.83	0.83	0.83	0.83	0.83	0.83
obf-FibIterat	obf-FibRecurs	0	0.47	0	0.47	0	0.48
obf-GCD1	obf-GCD2	0	0.4	0	0.4	0	0.42
obf-Virus	inf-GCD1	0	0.75	0	0.75	0	0.76
obf-Virus	inf-GCD2	0	0.74	0	0.74	0	0.75

Table 2
Test checking isomorphism and sub-isomorphism

The experiments confirm that when checking for sub-graph isomorphims we can capture more similarities. Indeed, the similarity between infected program and their original version can only be captured when considering sub-graph isomorphism, while the similarity between obfuscated variants of code can also be captured when considering isomorphism over the whole abstract CFGs. At the same time sub-graph isomorpism lead to global similarity values that are in general lower than those obtained when an isomorphism between the whole graphs is found, this depends on the definition of global-similarity that considers the local similarities over the entire graphs and not only over the size of the match that has been found.

It is not surprising that the results that we have obtained with the three abstraction are close to each other. This is because these abstractions share many common features: they all consider only assignment instructions and they perform similar abstractions on such instructions. In particular, the results of α_1 and α_2 are equal expect for one test (`GCD2` vs. `obf-GCD2`): in this case in the original program, a function was renamed through different assignment variables by the obfuscator and the first abstraction did not catch the binding to the same object (by recognizing the element firstly as a real variable and then as a function) while the second did. The experiments show that α_3 is more precise than the others in accordance with

our expectations. In only one case (`GCD1` vs. `inf-GCD1`), α_3 loses precision because the portion code that was not equal was a little greater than the one caught by the sub-graph isomorphism.

From the experiments that we have run (that are more than the ones presented in the above table) we were able to suggest to fix the threshold at `0.74`. If we use this threshold in the tests considered in Table 2 we can see that there are similarities that we are able to catch and others that we do not recognize. Moreover, wrt this threshold the three abstractions behave in the same way. The test performed confirmed that our prototype is insensitive to dead-code insertion, since the CFGs are extracted dynamically, and to the renaming of variables, functions, classes and methods, because of the basic block abstractions considered. It is insensitive also to code transposition because every basic block is considered as a set of instruction and similarity check based on sets can capture mixed code but it loses in precision because sets are not ordered. To a certain extent we are insensitive also to instruction substitution, because abstraction looks only for assignments and not for all possible instructions. On the other hand, this leads to catch programs with the same assignments but which could be different in semantics.

5 Conclusions and future works

The work done so far can be seen as a first step towards the development of a similarity tool parametric on the program property used to decide similarity, and it can be extended in many ways. We have implanted and considered only three basic block abstractions that are very similar to each other and that act on single instructions of basic block. A possible improvement would be to design abstractions that take into account either syntactic or semantic properties of the entire block. The overall similarity testing methodology that we have proposed works block-wise, in line with [7] but also with many existing tools for similarity analysis like [13,9]. In order to overcome this limitation we should consider properties of the entire CFGs and not only of the basic blocks. In this direction we plan to use some temporal logic to express properties of graphs. We plan to extend the similarity analysis also to the SFA representation of programs. We already have an algorithm for translating CFGs into SFAs. What we need is to develop abstractions of SFAs that take into account both the syntactic aspects of the predicates labeling the transitions and the semantics of predicates that determine what is recognized by the SFAs. This process requires a lot of work and is what we plan to do in the near future.

References

[1] A. Aiken. Moss: a system for detecting software plagiarism. https://theory.stanford.edu/~aiken/moss/

[2] C. Collberg and J. Nagra. Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. 1st edition, Addison-Wesley Professional, 2009.

[3] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proc. of the *4th ACM Symposium on Principles of Programming Languages (POPL 77)*, pages 238252. ACM Press, 1977.

[4] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In Proc. of the *6th ACM Symposium on Principles of Programming Languages (POPL 79)*, pages 269282. ACM Press, 1979.

[5] L. DAntoni and M. Veanes. Minimization of symbolic automata. In Proc. of the *41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pages 541554. ACM, 2014.

[6] M. Dalla Preda and R. Giacobazzi. Semantics-based code obfuscation by abstract interpretation. In *Journal of Computer Security*, num 6, vol 17, pages 855 - 908. 2009

[7] M. Dalla Preda, R. Giacobazzi, A. Lakhotia and I. Mastroeni. Abstract Symbolic Automata: Mixed syntactic/semantic similarity analysis of executables. In Proc. of the *42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015*. pages 329 - 341, 2015.

[8] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In Proc. of the *8th International Symposium on static Analysis (SAS)*, pages 40-56, Springer 2001.

[9] C. Krügel, E. Kirda, D. Mutz, W. K. Robertson and G. Vigna. Polymorphic Worm Detection Using Structural Information of Executables. In Proc. *Recent Advances in Intrusion Detection, 8th International Symposium, RAID 2005*. pages 207 - 226, Springer LNCS 3858, 2005.

[10] D. Bruschi, L. Martignoni and M. Monga. Detecting Self-mutating Malware Using Control-Flow Graph Matching. In Proc. *Detection of Intrusions and Malware & Vulnerability Assessment, Third International Conference, DIMVA 2006*, pages 129 -143, Springer LNCS 4046, 2006.

[11] D. Gao, M. Reiter and D. Song. BinHunt: Automatically Finding Semantic Differences in Binary Programs. in Proc. of the *10th International Conference on Information and Communications Security* pages 238 - 255, Springer-Verlag, 2008

[12] K. Jeonguk, S. Hyungjoon, K. Dongjin, J. Youn-Sik, C. Seong-je, P. Minkyu, H. Sangchul and K. Seong Baeg. Measuring Similarity of Android Applications via Reversing and K-gram Birthmarking. In Proc. of the *Research in Adaptive and Convergent Systems (RACS)*, pages 336 - 341, ACM 2013.

[13] A. Lakhotia, M. Dalla Preda and R. Giacobazzi. Fast location of similar code fragments using semantic 'Juice'. In Proc. of the *2nd Workshop on Program Protection and Reverse Engineering PPREW*, ACM 2013.

[14] I. Krsul and E. Spafford. Authorship analysis: Identifying the author of a program. Technical report CSD-TR-94-030. Computer science department, Purdue University 1994.

[15] J, Park, H. Kim, Y. Jeong, S. Han and M. Park. Effects of code obfuscation on Android app similarity analysis. In *Journal of wireless mobile networks, ubiquitous computing and dependable applications*, volume 8, number 4, pages 86 - 98. 2015.

[16] M. Polino, A. Scorti, F. Maggi and S. Zanero: Jackdaw: Towards Automatic Reverse Engineering of Large Datasets of Binaries. In Proc. of the *Detection of Intrusions and Malware & Vulnerability Assessment, Third International Conference, DIMVA 2015* pages 121-143, Springer LNCS 9148, 2015.

[17] S. Schleimer, D. Wilkerson and A. Aiken. Winnowing: local algorithms for document fingerprint. In Proc. of the *SIGMOD conference 2003*, ACM 2003.

[18] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjrner. Symbolic finite state transducers: algorithms and applications. In Proc. of the *ACM Symposium on Principles of Programming Languages POPL*, pages 137150. ACM, 2012.

[19] Zynamics. BinDiff3.2manual. http://www.zynamics.com/bindiff/manual/, 2004.

[20] *Trace or track Python statement execution*, https://docs.python.org/2/library/trace.html

[21] *NetworkX*, https://networkx.github.io/

[22] *Isomorphism in NetworkX*, https://networkx.github.io/documentation/latest/reference/algorithms.isomorphism.html

[23] L. P. Cordella, P. Foggia, C. Sansone, M. Vento, *A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs*, IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 26, no. 10, pp. 1367-1372, Oct., 2004

[24] *PyMinifier*, https://liftoff.github.io/pyminifier/pyminifier.html