Extracting malicious behaviours

Khanh Huu The Dam, Tayssir Touili

Extracting malicious behaviours

Khanh Huu The Dam* and Tayssir Touili

LIPN,
CNRS,
University Paris 13, France
Email: dam@lipn.univ-paris13.fr
Email: touili@lipn.univ-paris13.fr
*Corresponding author

Abstract: In recent years, the damage cost caused by malwares is huge. Thus, malware detection is a big challenge. The task of specifying malware takes a huge amount of time and engineering effort since it currently requires the manual study of the malicious code. Thus, in order to avoid the tedious manual analysis of malicious codes, this task has to be automatised. To this aim, we propose in this work to represent malicious behaviours using *extended API call graphs*, where nodes correspond to API function calls, edges specify the execution order between the API functions, and edge labels indicate the dependence relation between API functions parameters. We define new *static analysis techniques* that allow to extract such graphs from programs, and show how to automatically extract, from a set of malicious and benign programs, an extended API call graph that represents the malicious behaviours. Finally, we show how this graph can be used for malware detection. We implemented our techniques and obtained encouraging results: 95.66% of detection rate with 0% of false alarms.

Keywords: malware detection; static analysis; information extraction.

Reference to this paper should be made as follows: Dam, K.H.T. and Touili, T. (xxxx) 'Extracting malicious behaviours', *Int. J. Information and Computer Security*, Vol. x, No. x, pp.xxx–xxx.

Biographical notes: Khanh Huu The Dam obtained his PhD from the Paris 7 University in 2018. He is interested in program analysis, machine learning, deep learning and malware detection.

Tayssir Touili is a senior researcher in CNRS, France. She received her PhD from the Paris 7 University in 2003. In 2003–2004, she held her Research Fellow position in the Carnegie Mellon University, Pittsburgh, USA. She has more than 73 publications in several high-level, peer reviewed conferences and journals. Her research interests include software verification, binary code analysis, and malware detection.

This paper is a revised and expanded version of a paper entitled 'Precise extraction of malicious behaviors' presented at The 42nd IEEE International Conference on Computers, Software & Applications Staying Smarter in a Smartening World, Tokyo, Japan, 23–27 July 2019.

1 Introduction

1.1 Malware detection

Malware detection is nowadays a big challenge. Indeed, the damages caused by malwares in recent years is huge, e.g., in 2017, the global ransomware damage cost exceeds five billion dollars, by referring to Cyber Security Ventures (2017). However, most of industry anti-malware softwares are not robust enough because they are based on the signature matching technique. In this technique, a scanner will search for patterns in the form of binary sequences (called signatures) in the program. This highly depends on the database of malware signatures which are manually constructed by experts. If the scanner finds a signature in the database matching a new observed program, the scanned program will be declared as a malware. This signature matching approach can be easily evaded because the malicious code may be repacked or encrypted by using obfuscation techniques while keeping the same behaviours.

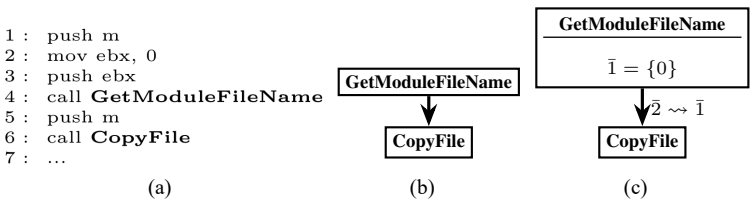
Emulation is another approach for malware detection where the behaviours are dynamically observed while running the program on an emulated environment. Although in the emulated environment one can capture the running behaviours of the program, it is hard to trigger the malicious behaviours in a short period since they may require a delay or only show up after user interaction.

To overcome the limitations of the above approaches, model checking was applied for malware detection (Christodorescu and Jha, 2003; Bergeron et al., 1999; Kinder et al., 2010; Song and Touili, 2013), since it allows to analyse the behaviours (not the syntax) of the program without executing it. However, this approach needs to take as input a formal specification of the malicious behaviours. Then, a model-checking algorithm is applied to detect whether a new program contains any of these behaviours or not. If it can find some of these behaviours in a given program, this latter is marked as malicious. Specifying the malicious behaviours requires a huge amount of engineering effort and an enormous amount of time, since it currently requires manual study of malicious codes. Thus, the main challenge is how to *automatise* this task of specifying malware to avoid the tedious manual analysis of malicious code and to be able to find a big number of malicious behaviours. To this aim, we need to

1 find an adequate formal representation of malicious behaviours

2 define analysis techniques to *automatically* discover such representations from malicious and benign programs.

Figure 1 A piece of assembly code of, (a) the behaviour self replication (b) its API call graph (c) its data dependence graph



1.2 Representation of malicious behaviours

It has been widely observed that malicious tasks are usually performed by calling sequences of API functions, since API functions allow to access the system and modify it. Thus, in previous works (Kinable and Kostakis, 2011; Kong and Yan, 2013; Xu et al., 2013; Dam and Touili, 2016), the malicious behaviours were characterised as API call graphs where nodes are API functions. Such graphs represent the execution order of API function calls in the program. For instance, let us look at a typical malicious behaviour implemented by the assembly code in Figure 1(a). This behaviour expresses a self replication in which the malware infects the system by copying itself to a new location. This is achieved by first calling the API function `GetModuleFileName` with 0 as first parameter and m as second parameter (parameters to a function in assembly are passed by pushing them onto the stack before a call to the function is made. The code in the called function later retrieves these parameters from the stack). This will store the file path into the memory address m. Then, CopyFile is called with m as first parameter. This allows to infect other files. To represent this behaviour, Kinable and Kostakis (2011), Kong and Yan (2013), Xu et al. (2013) and Dam and Touili (2016) use the API call graph in Figure 1(a) to express that calling `GetModuleFileName` is followed by a call to `CopyFile`. However, a program that contains this behaviour is malicious only if the API function `CopyFile` copies the file returned by `GetModuleFileName`. If `CopyFile` copies another file, the behaviour is not malicious. Thus, the above representation may lead to false alarms. To avoid this, we need to make the representation more precise and add the information that the returned parameter of `GetModuleFileName` should be the input argument of `CopyFile`. Therefore, we propose to use the extended API call graph in Figure 1(c), where the edge labelled by $\bar{2} \rightsquigarrow \bar{1}$ means that the second parameter of `GetModuleFileName` (which is its output) is given as first argument of `CopyFile`. We also need to ensure that `GetModuleFileName` is called with 0 as first parameter. Thus, we label the node `GetModuleFileName` with $\bar{1} = \{0\}$ to express that the first parameter of this call should be 0. Thus, we propose in this work to use *extended API call graphs* to represent malicious behaviours.

An extended API call graph is a directed graph whose nodes are API functions. An edge (f, f') expresses that there is a call to the API function f followed by a call to the API function f'. The annotation $\bar{i} \rightsquigarrow \bar{j}$ on the edge (f, f') means that the i^{th} parameter of function f and the j^{th} parameter of the function f' have a data dependence relation. It means that, in the program, either these two parameters depend on the same value or one parameter depends on the other, e.g., in Figure 1(c) the edge `(GetModuleFileName, CopyFile)` with label $\bar{2} \rightsquigarrow \bar{1}$ expresses that the second parameter ($\bar{2}$) of `GetModuleFileName` and the first parameter ($\bar{1}$) of `CopyFile` gets the same memory address m. A node f with the annotation $\bar{i} = \{c\}$ means that the i^{th} parameter of function f gets the value c, e.g., the node `GetModuleFileName` in Figure 1(c) is associated with the label $\bar{1} = \{0\}$. This graph specifies the execution order of the API function calls like the API call graph in Kinable and Kostakis (2011), Kong and Yan (2013), Xu et al. (2013) and Dam and Touili (2016). In addition, it records the links between the API functions' parameters.

In order to compute an extended API call graph of a given program, we need to compute the relation between the program's variables at the different control points. For instance if there is an instruction $x = y + z$ at control point n, we infer that at n,

x depends on y and z. Since in assembly, function parameters are passed by pushing them onto the stack before the call is made, to determine the relation between function arguments and the other variables of the program, we need to look into the program's stack. To this aim, we model the program (given as a binary code) by a pushdown system (PDS). A PDS can be seen as a finite automaton equipped with a stack. The PDS stack allows to mimic the program's stack. In order to be able to evaluate the dependence between the function arguments and the other variables of the program, we propose a new translation from binary code to PDSs different from the standard one (Song and Touili, 2016), where we push a pair (x, n) into the PDS stack if at control point n, the variable x is pushed (whereas in the standard translation, only the value of x at point n is pushed in the PDS stack). This allows to determine that this variable x in the stack comes from control point n, which enables to evaluate the relation it has with the other variables of the program. Then, to compute the relation between the different variables and function arguments of the program, we represent potentially infinite configurations of programs (PDSs) using finite automata, and we adapt the PDS $post^*$ saturation procedure of Esparza et al. (2000) in order to compute an annotation function from which we can compute the variable dependence function at each control point of the program. This allows to compute the relation between program variables and function arguments, and thus, to compute the extended API call graph of the program. Note that, as far as we know, this is the first time that a malicious behaviour representation that takes into account data dependence between variables is computed in a *static* way. All the other works that we are aware of use *dynamic* analysis for this purpose. Therefore, they cannot determine all the possible relations between variables and function arguments, since dynamic analysis allows to consider only a limited, finite number of paths of the program, and might skip the malicious paths and consider only the benign ones, since malicious behaviours might appear after some time, or after user interaction. The only other work we are aware of that tries to compute such malicious representations in a static way is Macedo and Touili (2013). However, the static computation of Macedo and Touili (2013) is not precise, as it states that two variables are related if they have the same value, not if they depend on each other.

Then, given a set of extended API call graphs that correspond to malwares and a set of extended API call graphs corresponding to benign programs, our goal is to *automatically* extract an extended API graph that corresponds to the malicious behaviours of the malwares. This *malicious* extended API graph is meant to represent the parts of the extended API call graphs of the malwares that correspond to the malicious behaviours. We should extract the subgraphs able to distinguish the malicious extended API call graphs from the benign ones. Therefore, our purpose is to isolate the few relevant subgraphs from the nonrelevant ones. This problem can be seen as an information retrieval (IR) problem, where the goal is to retrieve relevant items and reject nonrelevant ones. The IR community has been working on this problem for a long time. It has a large amount of experience on how to efficiently retrieve information. Thus, following Dam and Touili (2016), we adapt the knowledge and experience of the IR community to our malicious behaviour extraction problem. One of the most popular and efficient techniques in the IR community is the TFIDF scheme that computes the relevance of each item in the collection using the TFIDF weight. This weight is computed from the occurrences of terms in a document and their appearances in other documents. We adapt this approach that was mainly applied for text and image retrieval for malicious extended API graph extraction. For that, we associate to each node and

each edge in the extended API call graphs of the programs of the collection a weight. Higher weight implies higher relevance. Then, we compute the malicious extended API graphs by taking edges and nodes that have the highest weights.

We implement our approach and evaluate it on a dataset of 2,249 benign programs and 4,035 malicious programs collected from Vx Heaven (vxheaven.org) and VirusShare.com. We first applied our tool to automatically extract an extended malicious API graph from a set of 2,124 malwares and 1,009 benign programs. The obtained extended malicious API graph is then used for malware detection on a test set of 1,911 malwares and 1,240 benign programs. We obtained a detection rate of 95.6% and 0 false alarms, whereas in the approach of Dam and Touili (2016) based on API call graphs, false alarms can reach more than 10%. This shows the efficiency of our techniques and the importance of using *extended* API call graphs, rather than API call graphs. This paper is an extended version of the conference papers (Dam and Touili, 2018, 2016).

2 Related work

Schultz et al. (2001), Kolter and Maloof (2004), Gavrilut et al. (2009), Tahan et al. (2012) and Khammas et al. (2015) apply machine learning techniques for malware classification. All these works use either a vector of bits (Schultz et al., 2001; Gavrilut et al., 2009) or n-grams (Kolter and Maloof, 2004; Tahan et al., 2012; Khammas et al., 2015) to represent a program. Such vector models allow to record some chosen information from the program, they do not represent the program's behaviours. Thus they can easily be evaded by standard obfuscation techniques, whereas the representation of our extended API graph is more precise and represents the behaviour of programs via API calls and can thus resist to several obfuscation techniques.

Ravi and Manoharan (2012) uses sequences of API function calls to represent programs and learn malicious behaviours. Each program is represented by a sequence of API functions which are captured while executing the program. Similarly, Kruegel et al. (2003) also takes into account the system calls in the program. However, they only consider the length of the string arguments and the distribution of characters in the string arguments as features for their learning models. Rieck et al. (2008) uses as model a string that records the number of occurences of every function in the program's runs. Our model is more precise and more robust than these representations as it allows to take into account several API function sequences in the program while keeping the order of their execution. Moreover, Ravi and Manoharan (2012), Kruegel et al. (2003) and Rieck et al. (2008) use dynamic analysis to extract a program's representation. Our extended API graph extraction is done in a static way.

Dam and Touili (2016), Cheng et al. (2013), Santos et al. (2013), Shafiq et al. (2009), Canzanese et al. (2015), Ye et al. (2009), Khammas et al. (2015), Lin et al. (2015), Baldangombo et al. (2013), Masud et al. (2008) and Kapoor and Dhavale (2016) apply IR for malware detection. The TFIDF weighting scheme is used in Cheng et al. (2013) to compute the set of API functions that are relevant for malwares. This scheme is also used in Santos et al. (2013) to compute the set of relevant sequences of opcodes. Our malicious graph specifications are more robust since they take into account sequences of API function calls. As for Baldangombo et al. (2013), Lin et al. (2015), Masud et al. (2008), Ye et al. (2009), Khammas et al. (2015), Kapoor and Dhavale (2016), Shafiq et al. (2009) and Canzanese et al. (2015), they use IR techniques to reduce the

size of the program's representation by removing the irrelvant information. These works do not use IR techniques for malicious behaviour extraction. Dam and Touili (2016) applies the TFIDF weighting scheme to extract API call graphs as malicious behaviours. These graphs are less precise than our extended API call graphs, as they consider only the execution order between API functions, they do not consider neither the parameter values, nor the data dependence betweeen the functions' arguments. In Section 8, we give experimental evidence that shows that our approach is better than the one of Dam and Touili (2016).

Using a graph representation, Anderson et al. (2011) takes into account the order of execution of the different instructions of the programs (not only API function calls). Our extended API call graph representation is more robust. Indeed, considering all the instructions in the program makes the representation very sensitive to basic obfuscation techniques. Kinable and Kostakis (2011), Kong and Yan (2013) and Xu et al. (2013) use graphs where nodes are functions of the program (either API functions or any other function of the program). Such representations can easily be fooled by obfuscation techniques such as function renaming. Moreover, these approaches do not extract the malicious behaviours while we are able to extract malicious behaviours.

Christodorescu et al. (2007), Fredrikson et al. (2010), Macedo and Touili (2013), Elhadi et al. (2013) and Nikolopoulos and Polenakis (2016) represent programs using graphs similar to our extended API call graphs. However, Christodorescu et al. (2007), Fredrikson et al. (2010) and Elhadi et al. (2013) use dynamic analysis to compute the graphs, whereas our graph extraction is made statically. Dynamic analysis is not precise enough since it allows to consider only a finite number of paths of the program, whereas static analysis (as we do) is much more precise as it takes into account all programs paths. Macedo and Touili (2013) tries to compute the graphs corresponding to the malicious behaviours in a static way. However, the malicious behaviour representation and the static computation of Macedo and Touili (2013) are not precise, as they state that two variables are related if they have the same value, not if they depend on each other. Our representation is more precise since we take into account the data dependence relation between the arguments (not only their values). Indeed, it might be the case that two arguments have the same values in the program whereas they do not depend on each other. Our technique allows to perform static analysis to compute the data dependence relation in the program without comparing the values of the arguments. On the other hand, Christodorescu et al. (2007), Fredrikson et al. (2010) and Macedo and Touili (2013) use graph mining algorithms to compute the subgraphs that belong to malwares and not to benign programs and they assume that these correspond to malicious behaviours. We do not make such assumption as two malwares may not have any common subgraphs.

Nikolopoulos and Polenakis (2016) use a kind of an API call graph, where each node corresponds to a group of API function calls. Our graphs are more precise since we do not group API functions together. Moreover, Nikolopoulos and Polenakis (2016) uses dynamic analysis to extract graphs, whereas our techniques are static.

As for Bhatkar et al. (2006), they consider data relations between functions' arguments to characterise malicious behaviours. This approach is less precise than ours since it does not take into account API function names. Moreover, Bhatkar et al. (2006) is based on dynamic analysis to compute such data flow relation, whereas we compute the data relation in a static way.

Outline: In Section 3, we present our new translation from binary programs to PDSs. Section 4 defines the data dependence relation that gives the link between the different variables and function parameters of the program. We present our algorithm to compute this data dependence relation in Section 5. Extended API call graphs are defined in Section 6. We present our TFIDF algorithm to automatically extract an extended API call graph from a set of malicious and benign programs in Section 7. Section 8 reports our experimental results. Section 9 presents some malicious behaviours automatically extracted by our tool and Section 10 concludes.

3 Modelling binary programs

In this section, we show how to build a PDS from a binary program. We suppose we are given an oracle $\mathcal{O}$ that extracts from the binary program a control flow graph (CFG) equipped with information about the values of the registers and the memory locations at each control point of the program. In our implementation, we use Jakstab (Kinder and Veith, 2008) and IDA Pro (Eagle, 2011) to get this oracle. Our translation from a binary code to a PDS is different from the standard one (Song and Touili, 2016): we push a pair (x, n) into the PDS stack if at control point n, the variable x is pushed [whereas in the standard translation of Song and Touili (2016), the value of x at control point n is pushed in the PDS stack]. This allows to determine that this variable x in the stack comes from control point n, which enables to evaluate the relation it has with the other variables of the program. Note that pushing such a pair (x, n) is *crucial* to determine the data dependence relation between the function parameters of the program. Indeed, in assembly, function parameters are passed by pushing them onto the stack before the call is made. Thus, to determine the relation between function arguments and the other variables of the program, we need to retrieve these arguments from the stack, and determine on what variables do they depend. This can be achieved by pushing pairs of the form (x, n) onto the PDS stack.

3.1 Control flow graphs

A CFG is a tuple $G = (N, I, E)$, where N is a finite set of nodes, I is a finite set of assembly instructions in a program, and $E : N \times I \times N$ is a finite set of edges. Each node corresponds to a control point (a location) in the program. Each edge connects two control points and is associated with an assembly instruction. An edge (n, i, n') in E expresses that in the program, the control point n is followed by the control point n' and is associated with the instruction i. We write $n \xrightarrow{i} n'$ to express that i is an instruction from control point n to control point n', i.e., that $(n, i, n') \in E$.

Let p_0 be the entry point of the program. If there exist edges (p_0, i_1, p_1), $(p_1, i_2, p_2) \dots (p_{k-1}, i_k, p_k)$ in the CFG, then $\rho = i_1, i_2, \dots, i_k$ is a path leading from p_0 to p_k, we write $p_0 \xrightarrow{i_1} p_1 \xrightarrow{i_2} p_2 \dots p_{k-1} \xrightarrow{i_k} p_k$ or $p_0 \rightarrow_\rho p_k$.

Let X be the set of data region names used in the program. From now on, we call elements in X *variables*. Given a binary program, the oracle $\mathcal{O}$ computes a corresponding CFG G equipped with information about the values of the variables at each control point of the program: for every control point n and every variable x of the program, $\mathcal{O}(n)(x)$ is an overapproximation of the possible values of variable x at control point n.

3.2 Pushdown systems

A *PDS* (Bouajjani et al., 1997) is a tuple $\mathcal{P} = (P, \Gamma, \Delta)$, where P is a finite set of control locations, Γ is the stack alphabet, $\Delta \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ is a finite set of transition rules.

A configuration $\langle p, \omega \rangle$ of $\mathcal{P}$ is an element of $P \times \Gamma^*$. We write $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle$ instead of $((p, \gamma), (q, \omega)) \in \Delta$. The successor relation $\rightsquigarrow_{\mathcal{P}} \subseteq (P \times \Gamma^*) \times (P \times \Gamma^*)$ is defined as follows: if $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle$, then $\langle p, \gamma\omega' \rangle \rightsquigarrow_{\mathcal{P}} \langle q, \omega\omega' \rangle$ for every $\omega' \in \Gamma^*$. A *path* of the PDS is a sequence of configurations $c_1, c_2, ...$ such that c_{i+1} is an *immediate successor* of the configuration c_i, i.e., $c_i \rightsquigarrow_{\mathcal{P}} c_{i+1}$, for every $i \geq 1$. Let $\rightsquigarrow^*_{\mathcal{P}} \subseteq (P \times \Gamma^*) \times (P \times \Gamma^*)$ be the transitive and reflexive relation of $\rightsquigarrow_{\mathcal{P}}$ such that for every $c, c' \in P \times \Gamma^*$, $c \rightsquigarrow^*_{\mathcal{P}} c$, and $c \rightsquigarrow^*_{\mathcal{P}} c'$ iff there exists $c'' \in P \times \Gamma^*$: $c \rightsquigarrow_{\mathcal{P}} c''$ and $c'' \rightsquigarrow^*_{\mathcal{P}} c'$. For every set of configurations $C \subseteq 2^{P \times \Gamma^*}$, let $post^*(C) = \{c \in P \times \Gamma^* \mid \exists c' \in C : c' \rightsquigarrow^*_{\mathcal{P}} c\}$ be its set of successors.

To finitely represent (potentially) infinite sets of configurations of PDSs, we use multi-automata: given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$, a *multi-automaton* (MA) is a tuple $\mathcal{M} = (Q, \Gamma, \delta, Q_0, Q_F)$, where Q is a finite set of states, $\delta : Q \times \Gamma \times Q$ is a finite set of transition rules, $Q_0 \subseteq Q$ is a set of initial states corresponding to the control locations P, $Q_F \subseteq Q$ is a finite set of final states. Let $\rightarrow_\delta : Q \times \Gamma^* \times Q$ be the transition relation such that for every $q \in Q$: $q \xrightarrow{\epsilon}_\delta q$ and $q \xrightarrow{\gamma\omega}_\delta q'$ if $(q, \gamma, q'') \in \delta$ and $q'' \xrightarrow{\omega}_\delta q'$. A configuration $\langle p, \omega \rangle \in P \times \Gamma^*$ is accepted by $\mathcal{M}$ iff $p \xrightarrow{\omega}_\delta q$ for some $q \in Q_F$. A set of configuration $\mathcal{C} \subseteq P \times \Gamma^*$ is *regular* iff there exists a MA $\mathcal{M}$ such that $\mathcal{M}$ exactly accepts the set of configurations $\mathcal{C}$. Let $L(\mathcal{M})$ be the set of configurations accepted by $\mathcal{M}$.

3.3 From CFGs to PDSs

Let $\mathbf{X}$ be the set of data region names used in the program. From now on, we call elements in $\mathbf{X}$ *variables*. Given a binary program, the oracle $\mathcal{O}$ computes a corresponding CFG G equipped with information about the values of the variables at each control point of the program: for every control point n and every variable x of the program, $\mathcal{O}(n)(x)$ is an overapproximation of the possible values of variable x at control point n.

We define the PDS $\mathcal{P} = (P, \Gamma, \Delta)$ corresponding to the CFG $G = (N, I, E)$ as follows. We suppose w.l.o.g. that initially, $\mathcal{P}$ has $\sharp$ in its stack.

- The set of control points P is the set of nodes $N \cup N'$ where N' is a finite set,
- Γ is the smallest subset of $\mathbf{X} \times N \cup \{\tau\} \cup \{\sharp\} \cup \mathbb{Z} \times \{\top\}$, where $\mathbf{X}$ is the set of variables of the program and $\mathbb{Z}$ is the set of integers, satisfying the following:

  a If $n \xrightarrow{push\ x} n'$, where x is a variable, then $(x, n) \in \Gamma$ (x is pushed into the stack at control point n).

  b If $n \xrightarrow{push\ c} n'$, where c is a constant, then $(c, \top) \in \Gamma$ (the constant c is pushed into the stack).

  c If $n \xrightarrow{call\ f} n'$ where $n' \in N$ then $(n', \top) \in \Gamma$ (the return address n' is pushed into the stack).

- The set of rules Δ contains transition rules that mimic the program's instructions: for every $(n, i, n') \in E$ and $\gamma \in \Gamma$:

  α_1 If i is $pop\ x$, add the transition rule $\langle n, \gamma\rangle\ \langle n', \epsilon\rangle \in \Delta$. This rule pops the topmost symbol from the stack and moves the program's control point to n'.

  α_2

  a If i is $push\ x$, where x is a variable, add the transition rule $\langle n, \gamma\rangle\ \langle n', (x, n)\gamma\rangle \in \Delta$: (x, n) is pushed to the stack. This allows to record that x was pushed at control point n.

  b If i is $push\ c$, where c is a constant, add the transition rule $\langle n, \gamma\rangle\ \langle n', (c, \top)\gamma\rangle \in \Delta$: $(c, \top)$ is pushed to express that a constant does not depend on any other value.

  α_3 If i is $sub\ esp, c$. Here, c is the number in bytes by which the stack pointer esp is decremented. This amounts to push $k = c/4$ symbols into the stack. Then, we would like to add the transition rule $\langle n, \gamma\rangle\ \langle n', \tau^k\gamma\rangle \in \Delta$: we push k τ's because we do not have any information about the content of the memory region above the stack. For technical reasons, we require that the rules of the PDS are of the form $\langle p, \gamma\rangle\ \langle p', w\rangle$, where $|w| \leq 2$. This is needed in the saturation procedure, but this is not a restriction, since any PDS can be transformed into a PDS that satisfies this constraint (Schwoon, 2002). Thus, instead of adding the above rule to push k τ's, we add the following rules to Δ: $\langle n, \gamma\rangle\ \langle n'_1, \tau\gamma\rangle$, for every $1 < i < k$, $\langle n'_{i-1}, \tau\rangle\ \langle n'_i, \tau\tau\rangle$, and $\langle n'_{k-1}, \tau\rangle\ \langle n', \tau\tau\rangle$, where $n'_i \in N'$, for $1 \leq i \leq k$.

  α_4 If i is $jmp\ x$, we add the transition rule $\langle n, \gamma\rangle\ \langle n_x, \gamma\rangle \in \Delta$ for every $n_x \in \mathcal{O}(n)(x)$, where $\mathcal{O}(n)(x)$ is the set of possible values of x at control point n. This rule moves the program's control point to all the possible addresses that can be values of x.

  α_5 If i is $cjump\ x$ where $cjump$ denotes a conditional jump instruction (je, jg, jz, etc.), we add two transition rules $\langle n, \gamma\rangle\ \langle n_x, \gamma\rangle \in \Delta$ for every $n_x \in \mathcal{O}(n)(x)$ where $\mathcal{O}(n)(x)$ is the set of possible values of x at control point n, and $\langle n, \gamma\rangle\ \langle n', \gamma\rangle \in \Delta$. The first rule moves the program's control point to all possible control points n_x (case where the condition is satisfied). The second rule moves the program's control point to n' (case where the condition is not satisfied).

  α_6 If i is $call\ f$ is a call to a function f. Let e_f be the entry point of the function f. We add the transition rule $\langle n, \gamma\rangle\ \langle e_f, (n', \top)\gamma\rangle \in \Delta$. This rule moves the program's control point to the entry point e_f and pushes $(n', \top)$: n' is the return address of the call. $\top$ expresses that it does not depend on other variables of the program since it is a control point.

  α_7 If i is ret. Let $addr$ be the return address of the function containing this return. We add the transition rules $\langle n, \gamma\rangle\ \langle addr, \epsilon\rangle \in \Delta$, for γ of the form $\gamma = (addr, \top)$, corresponding to a return address. As for $\gamma = (x, n'')$ where x is a variable pushed into the stack at n'', for every $addr' \in \mathcal{O}(n'')(x)$, we add the transition rules $\langle n, \gamma\rangle\ \langle addr', \epsilon\rangle \in \Delta$. These rules move the

program's control point to the return address of the function call and pop the topmost symbol from the stack.

α_8 If i is $add\ esp, c$. Here, c is the number in bytes by which the stack pointer esp is incremented. This amounts to pop $k = c/4$ symbols from the stack. Then, we add to Δ the transition rules $\langle n, \gamma\rangle\ \langle n'_1, \epsilon\rangle$, $\langle n'_{i-1}, \gamma\rangle\ \langle n'_i, \epsilon\rangle$ for every $1 < i < k$ and $\langle n'_{k-1}, \gamma\rangle\ \langle n', \epsilon\rangle$. These rules move the program's control point from n to n' and pop the k topmost symbols from the stack.

α_9 If i is any instruction which does not change the stack, we add the transition rules $\langle n, \gamma\rangle\ \langle n', \gamma\rangle \in \Delta$. This rule move the program's control point from n to n' without changing the stack.

4 The data dependence relation

As mentioned previously, to be able to compute the extended API call graph of a program, we first need to determine the dependence between variables and function arguments at the different control points of the program. To this aim, we define in this section a *data dependence relation* that evaluates such variable dependence.

From now on, let us fix $G = (N, I, E)$ as the CFG of the program and $\mathcal{P} = (P, \Gamma, \Delta)$ its corresponding PDS (as described in Subsection 3.3). Let $\mathbb{Z}$ be the set of integers, $\mathbf{X} = \mathbf{X}_{global} \cup \mathbf{X}_{local}$ be the program's set of variables, where $\mathbf{X}_{global}$ is the set of variables that are used in the whole program, and $\mathbf{X}_{local}$ is the set of local variables of the program's functions that are used only in the scope of functions (parameters of functions are local variables, while registers are global variables). The data dependence function is defined as follows: $\mathbf{Dep} : N \times X \rightarrow 2^{(X\times N)\cup(\mathbb{Z}\times\{\top\})}$, s.t.:

- $(y, p') \in \mathbf{Dep}(p, x)$ means that the variable x at the location p depends on the variable y at the location p', i.e., there exists a path from control point p' to control point p on which the value of x at p depends on the value of y at p', i.e., y is defined (assigned a value) at point p', and from p' to p, there is no definition of y, i.e., y is not assigned any other value from p' to p.
- $(c, \top) \in \mathbf{Dep}(p, x)$ means that the variable x at the location p depends on the constant value c. $\top$ in the pair $(c, \top)$ indicates that c is a constant value in the program.

We assume that there is no instruction leading to the entry point p_0 of the program. The data dependence function of any variable x at p_0 is $\mathbf{Dep}(p_0, x) = \emptyset$, since initially the variable x does not depend on any other variable.

Let $\rho = i_1, i_2, \ldots, i_k$ be a path of instructions leading to the location p_k from the entry point p_0: $p_0 \xrightarrow{i_1} p_1 \xrightarrow{i_2} p_2 \ldots p_{k-1} \xrightarrow{i_k} p_k$. We define $Dep_\rho(p_k, x)$, the data dependence function of x at the location p_k on the path ρ as follows:

- Initially $Dep_\rho(p_0, x) = \emptyset$.
- If i_k is an assignment of x to c, the variable x depends on the constant c on this path: $Dep_\rho(p_k, x) = \{(c, \top)\}$, and for every variable $y \in \mathbf{X} \setminus \{x\}$, $Dep_\rho(p_k, y) = Dep_\rho(p_{k-1}, y)$.

- If i_k is an assignment of x to an expression $exp(y_1 \dots y_m)$ (this denotes an expression that uses the variables $y_1, \dots, y_m$), the variable x is defined at p_{k-1} and depends on every variable y_j in this expression exp:
  $Dep_\rho(p_k, x) = \{(x, p_{k-1})\} \cup \{(y_j, p_{l_j}) | 1 \leq j \leq m$ such that y_j is defined at location p_{l_j}, $1 \leq l_j \leq k-1$ and there is no assignment to y_j between p_{l_j} and p_{k-1} on the path $\rho\}$. Moreover, for every variable $y \in \mathbf{X} \setminus \{x\}$,
  $Dep_\rho(p_k, y) = Dep_\rho(p_{k-1}, y)$.
- If i_k is $pop\ x$, the variable x is defined at p_{k-1} and depends on the value on the top of the program's stack.

  There are two cases:

  1 If the value on the top of the stack was pushed by an instruction of the form $push\ y$ at control point p_l, $1 \leq l \leq k-1$, for a variable y, then $Dep_\rho(p_k, x) = Dep_\rho(p_l, y)$. This case corresponds to the situation where the topmost symbol of the PDS $\mathcal{P}$ stack corresponding to this execution is (y, p_l).

  2 If the value on the top of the stack was pushed by an instruction of the form $push\ c$ at control point p_l, $1 \leq l \leq k-1$, for a constant c, then $Dep_\rho(p_k, x) = \{(c, \top)\}$. This case corresponds to the situation where the topmost symbol of the PDS $\mathcal{P}$ stack corresponding to this execution is $(c, \top)$.

  Moreover, $Dep_\rho(p_k, y) = Dep_\rho(p_{k-1}, y)$, for every variable $y \in \mathbf{X} \setminus \{x\}$.
- If i_k is $call\ f(v_1 \dots v_m)$. The location p_k is the return address of the function call. Since the call to f is made at p_{k-1}, the program will execute the function f first, i.e., will move the control point to the entry point e_f of the function f, and then after the execution of the function f, by the return statement, the program will move the control point from the exit point x_f of the function f to the return address p_k of this call. Since the arguments in the call to f are pushed into the stack, the parameters of the function f depend on the corresponding arguments in the stack, i.e., the topmost m values in the stack correspond to these arguments. At the entry point e_f, every parameter v_h $(1 \leq h \leq m)$ of the function f depends on the corresponding arguments on the stack. There are two cases for every argument v_h, $(1 \leq h \leq m)$:

  1 If the h^{th} element on top of the stack was pushed by an instruction of the form $push\ y_h$ at control point p_l, $1 \leq l \leq k-1$, for a variable y_h, then $Dep_\rho(e_f, v_h) = Dep_\rho(p_l, y_h)$. This case corresponds to the situation where the h^{th} symbol of the PDS $\mathcal{P}$ stack corresponding to this execution is (y_h, p_l).

  2 If the h^{th} element on the top of the stack was pushed by an instruction of the form $push\ c$ at control point p_l, $1 \leq l \leq k-1$, for a constant c, then $Dep_\rho(e_f, v_h) = \{(c, \top)\}$. This case corresponds to the situation where the h^{th} symbol of the PDS $\mathcal{P}$ stack corresponding to this execution is $(c, \top)$.

  Moreover, $Dep_\rho(e_f, y) = Dep_\rho(p_{k-1}, y)$, for every variable $y \in \mathbf{X} \setminus \{v_h\}_{1 \leq h \leq m}$.

  At the return address p_k (the return statement at x_f moves the program's control point to p_k),

1 $Dep_\rho(p_k, y) = Dep_\rho(x_f, y)$ for every global variable $y \in \mathbf{X}_{global}$, since the global variable y can be changed inside the function f.

2 $Dep_\rho(p_k, y) = Dep_\rho(p_{k-1}, y)$ for every local variable of the caller $y \in \mathbf{X}_{local}$, since a local variable y of the caller does not change in the called function f.

- For the other cases where the variable x is not changed by the instruction i_k, $Dep_\rho(p_k, x) = Dep_\rho(p_{k-1}, x)$.

Let $x \in \mathbf{X}$ be a variable and p be a location in the program. Let ρ_j $(1 \leq j \leq s)$ be all the paths leading from the entry point p_0 to p in the program ($p_0 \rightarrow_{\rho_j} p$). The data dependence function of variable x at the location p is defined as follows:

$$\mathbf{Dep}(p, x) = \bigcup_{1 \leq j \leq s} Dep_{\rho_j}(p, x).$$

5 Computing the data dependence relation

As described in Subsection 3.2, we use regular languages and multi-automata (MA) to describe potentially infinite sets of configurations of PDSs. Let $\mathcal{A} = (Q, \Gamma, \delta, Q_0, Q_F)$ be a MA of the PDS $\mathcal{P} = (P, \Gamma, \Delta)$, where $Q = P \cup \{q_F\}$, $Q_0 = \{p_0\}$ (p_0 is the entry point of the program), $Q_F = \{q_F\}$ and $\delta = \{(p_0, \sharp, q_F)\}$. Initially, the stack is empty (it only contains the initial special stack symbol $\sharp$). $\mathcal{A}$ accepts the initial configuration of the PDS $\langle p_0, \sharp \rangle$. We can compute an MA $\mathcal{A}' = (Q', \Gamma, \delta', Q_0, Q_F)$ that accepts $post^*(\langle p_0, \sharp \rangle)$. $\mathcal{A}'$ is obtained from $\mathcal{A}$ by the following procedure of Esparza et al. (2000): Initially, $\delta' = \delta$.

- For every transition rule $\langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma \rangle \in \Delta$, we add a new state $p'_{\gamma'}$, and a new transition rule $(p', \gamma', p'_{\gamma'})$ into δ'.
- We add new transitions into δ' by the following saturation rules: For every $(p, \gamma, q) \in \delta'$:

  1 If $\langle p, \gamma \rangle$ $\langle p', \gamma'\gamma'' \rangle \in \Delta$, add the new transition $(p'_{\gamma'}, \gamma'', q)$ into δ'.

  2 If $\langle p, \gamma \rangle$ $\langle p', \gamma' \rangle \in \Delta$, add the new transition (p', γ', q) into δ'.

  3 If $\langle p, \gamma \rangle$ $\langle p', \epsilon \rangle \in \Delta$, then for every transition $(q, \gamma', q') \in \delta'$, add the new transition (p', γ', q') into δ'.

We will annotate the transitions δ' of $\mathcal{A}'$ by an annotation function θ, s.t. for every control point p, and every transition rule $t = (p, \gamma, q)$ of $\mathcal{A}'$, for every variable $x \in X$, $(x', p') \in \theta(t)(x)$ (resp. $(c, \top) \in \theta(t)(x)$) expresses that there exists $w \in \Gamma^*$, $q \xrightarrow{w}_{\delta'} q_F$, such that in at least one of the paths leading from the initial configuration $\langle p_0, \sharp \rangle$ to $\langle p, \gamma w \rangle$, the value of the variable x at the configuration $\langle p, \gamma w \rangle$ depends on the value of the variable x' at the control point p' (resp. depends on the constant value c).

For every t in δ', let then $\theta(t) : X \rightarrow 2^{(X \times N) \cup (\mathbb{Z} \times \{\top\})}$ be a kind of dependence function. We compute these functions according to the following rules:

β_0 Initially, for every transition rule $t = (p, \gamma, q) \in \delta'$, let $\theta(t)(y) = \emptyset$ for every $y \in \mathbf{X}$.

β_1 If $\langle p, \gamma\rangle\ \langle p', \gamma'\rangle \in \Delta$. Then, by the above saturation procedure, for every $t = (p, \gamma, q)$ in δ', $t' = (p', \gamma', q)$ is also in δ'. Let i be the program instruction corresponding to this rule (i is an instruction from control point p to p': $p \xrightarrow{i} p'$). There are two cases, depending on whether i is an assignment or not:

- $\beta_{1.0}$ If i is not an assignment, then $\theta(t') := \theta(t) \cup \theta(t')$.
- $\beta_{1.1}$ If i is an assignment of the variable $y \in \mathbf{X}$, then $\theta(t')$ is computed as follows:
  - $\beta_{1.1.1}$ If i is an assignment of y to a constant value such that $y := c$, e.g., $mov\ y, c$ or $lea\ y, c$, etc. then $\theta(t')(y) := \theta(t')(y) \cup \{(c, \top)\}$.
  - $\beta_{1.1.2}$ If i is an assignment to an expression exp such that $y := exp$, e.g., $mov\ y, y'$ or $add\ y, y'$, etc. then $\theta(t')(y) := \theta(t')(y) \cup \{(y, p)\} \cup (\bigcup_{y' \in \mathrm{Var}(exp)} \theta(t)(y'))$, where $\mathrm{Var}(exp)$ is the set of variables used in the expression exp.

β_2 If $\langle p, \gamma\rangle\ \langle p', \gamma'\gamma''\rangle \in \Delta$ is a rule corresponding to an instruction i of the form $push\ y$ or $sub\ esp, x$ (where $p \xrightarrow{i} p'$). Then, by the saturation procedure, for every $t = (p, \gamma, q) \in \delta'$, $t' = (p', \gamma', p'_{\gamma'})$ and $t'' = (p'_{\gamma'}, \gamma'', q)$ are in δ'. Then $\theta(t')$ and $\theta(t'')$ are computed as follows:

- $\beta_{2.0}$ For every $y \in \mathbf{X}$, $\theta(t')(y) := \theta(t')(y) \cup \theta(t)(y)$.
- $\beta_{2.1}$ For every $y \in \mathbf{X}$, $\theta(t'')(y) := \theta(t'')(y) \cup \theta(t)(y)$.

β_3 If $\langle p, \gamma\rangle\ \langle e_f, \gamma'\gamma\rangle \in \Delta$ is a rule corresponding to an instruction i of the form $call\ f(v_1 \dots v_m)$ (where $p \xrightarrow{i} p'$). Then, by the saturation procedure, for every $t = (p, \gamma, q) \in \delta'$, $t' = (e_f, \gamma', e_{f_{\gamma'}})$ and $t'' = (e_{f_{\gamma'}}, \gamma, q)$ are in δ' where $\gamma' = (p', \top)$ (rule α_6). f has m arguments that should be taken from the stack. Let then d be the number of different prefixes of length m of accepting paths in the MA $\mathcal{A}'$ starting from the transition t. Let $\gamma_1^j, \cdots, \gamma_m^j$, for $j, 1 \le j \le d$, be such prefixes ($\gamma_1^j = \gamma$ for every $j, 1 \le j \le d$). (This means that for the path j in $\mathcal{A}'$, γ_k^j is the k^{th} symbol on the stack, i.e., the k^{th} argument of f, for $k, 1 \le k \le m$). Then, γ_k^j is either of the form (p_k^j, y_k^j), where $y_k^j \in X$ is a variable and $p_k^j \in P$ is a control point of the program; or of the form $(c_k^j, \top)$, where c_k^j is a constant. If γ_k^j is of the form (p_k^j, y_k^j), $1 \le k \le m$, $1 \le j \le d$, let $t_{s_k^j}$, $1 \le s_k^j \le h_k^j$ be all transitions in $\mathcal{A}'$ of the form $t_{s_k^j} = (p_k^j, \gamma_{s_k^j}, q_{s_k^j})$, i.e., be all transitions outgoing from p_k^j. Then $\theta(t')$ and $\theta(t'')$ are computed as follows.

- $\beta_{3.0}$ For $1 \le k \le m$:

$$\theta(t')(v_k) := \theta(t')(v_k) \cup \bigcup_{1 \le j \le d} \left(\bigcup_{\gamma_k^j = (p_k^j, y_k^j)} \left(\bigcup_{1 \le s_k^j \le h_k^j} \theta(t_{s_k^j})(y_k^j) \right) \right)$$

$$\cup \bigcup_{\gamma_k^j=(c_k^j,\top)} \{(c_k^j,\top)\}\Bigg)$$

$\beta_{3.1}$ For every $y \in \mathbf{X}_{global} \setminus \{v_k\}_{1\leq k\leq m}$, $\theta(t')(y) := \theta(t')(y) \cup \theta(t)(y)$.

$\beta_{3.2}$ For every $y \in \mathbf{X}_{local}$, $\theta(t'')(y) := \theta(t'')(y) \cup \theta(t)(y)$.

β_4 If $\langle x_f, \gamma\rangle \ \langle addr, \epsilon\rangle \in \Delta$ is a rule corresponding to the return instruction ret of the function f (α_7). Then, by the saturation procedure, for every $t^0 = (x_f, \gamma, q)$ and $t^1 = (q, \gamma', q')$ in δ', $t^2 = (addr, \gamma', q')$ is in δ'. $\theta(t^2)$ is computed as follows:

$\beta_{4.0}$ For every $y \in \mathbf{X}_{global}$, $\theta(t^2)(y) := \theta(t^2)(y) \cup \theta(t^0)(y)$.

$\beta_{4.1}$ For every $y \in \mathbf{X}_{local}$, $\theta(t^2)(y) := \theta(t^2)(y) \cup \theta(t^1)(y)$.

β_5 If $\langle p, \gamma\rangle \ \langle p', \epsilon\rangle \in \Delta$ is a rule corresponding to an instruction i of the form $pop\ y$ (where $p \xrightarrow{i} p'$). Then, by the saturation procedure, for every $t = (p, \gamma, q)$ and $t' = (q, \gamma', q')$ in δ', $t'' = (p', \gamma', q')$ is in δ'. $\theta(t'')$ is computed as follows:

$\beta_{5.0}$ For every $y' \in \mathbf{X} \setminus \{y\}$, $\theta(t'')(y') := \theta(t'')(y') \cup \theta(t)(y')$.

$\beta_{5.1}$ If γ is of the form $(c, \top)$, where c is a constant, then $\theta(t'')(y) := \theta(t'')(y) \cup \{(c, \top)\}$. Otherwise, if γ is of the form (p'', y') where $y' \in X$ is a variable and $p'' \in P$ is a control point of the program, then let t_k, $1 \leq k \leq j$ be all transitions in $\mathcal{A}'$ of the form $t_k = (p'', \gamma_k, q_k)$, i.e., t_k are all the outgoing transitions from p'' (this means that γ_k is a possible topmost stack symbol at p''). Then

$$\theta(t'')(y) := \theta(t'')(y) \cup \bigcup_{1\leq k\leq j} \Bigg(\bigcup_{\gamma_k=(p_k,y_k)} \theta(t_k)(y') \cup \bigcup_{\gamma_k=(c_k,\top)} \{(c_k,\top)\} \Bigg).$$

β_6 If $\langle p, \gamma\rangle \ \langle p', \epsilon\rangle \in \Delta$ is a rule corresponding to an instruction i of the form $add\ esp, x$. Then, by the saturation procedure, for every $t = (p, \gamma, q)$ and $t' = (q, \gamma', q')$ in δ', $t'' = (p', \gamma', q')$ is in δ'. $\theta(t'')$ is computed as follows: for every $y' \in \mathbf{X}$, $\theta(t'')(y') := \theta(t'')(y') \cup \theta(t)(y')$.

5.1 Intuition

Let us give the intuition behind the above rules. Item β_0 initialises the θ of all the transitions with the emptyset. Then, θ can be iteratively computed by applying items $\beta_1, \ldots, \beta_6$. The process terminates when for every transition t, $\theta(t)$ cannot be modified anymore.

Item β_1 deals with the case where there exist a transition rule $\langle p, \gamma\rangle \ \langle p', \gamma'\rangle$ in the PDS and $t = (p, \gamma, q)$ in $\mathcal{A}'$. By the saturation procedure described above, $t' = (p', \gamma', q)$

is in $\mathcal{A}'$. There are different cases depending on the nature of the instruction i of the program that corresponds to the PDS rule $\langle p, \gamma\rangle\ \langle p', \gamma'\rangle$:

- If i is not an assignment, then the dependence of the variables at p' is the same as their dependence at p. This is expressed by item $\beta_{1.0}$. We write $\theta(t') := \theta(t) \cup \theta(t')$, to express that the new value of $\theta(t')$ is equal to the old value of $\theta(t')$ (dependence of variables at p') union $\theta(t)$ (dependence of variables at p).
- If i is an assignment of y to a constant value such that $y := c$, then at p', y depends on the constant c. Thus, item $\beta_{1.1.1}$ adds $(c, \top)$ to $\theta(t')(y)$.
- If i is an assignment to an expression exp such that $y := exp$, then at p', y depends on the values of y' at p, for every variables y' used in the expression exp. Thus, item $\beta_{1.1.2}$ adds $\bigcup_{y' \in \text{Var}(exp)} \theta(t)(y')$ to $\theta(t')(y)$. Since y is defined at p, item $\beta_{1.1.2}$ adds also (y, p) to $\theta(t')(y)$.

Item β_2 expresses that a push instruction does not change the dependence of variables: if the instruction i from p to p' is a push, then for every variable y, the dependence of y at p' is the same as its dependence at p.

Item β_3 handles the case where there is a transition rule $\langle p, \gamma\rangle\ \langle e_f, \gamma'\gamma\rangle$ corresponding to an instruction i of the form $call\ f(v_1 \dots v_m)$ (where $p \xrightarrow{i} p'$). In this call, f has m arguments that should be taken from the stack. The γ_k^js in item β_3 are the different stack symbols such that for $j, 1 \leq j \leq d$, $\gamma_1^j \cdots \gamma_m^j$ is one of the possible prefixes of size m of the stack content at point p (there are d possible such prefixes). Since arguments to functions are passed through the stack, for every possible stack content (every possible path) j, the value of the parameter v_k, $1 \leq k \leq m$, is taken from γ_k^j. There are two cases that are expressed by item $\beta_{3.0}$:

1 If γ_k^j is of the from (p_k^j, y_k^j), where $y_k^j \in X$ is a variable and $p_k^j \in P$ is a control point of the program, then v_k depends on the value of y_k^j at point p_k^j (remember, the stack symbol (p_k^j, y_k^j) expresses that y_k^j was pushed into the stack at control point p_k^j). Thus, we need to determine the dependence of variable y_k^j at control point p_k^j. This dependence is determined by the θ of the transitions that are outgoing from p_k^j. Therefore, we add $\bigcup_{\gamma_k^j = (p_k^j, y_k^j)} \left(\bigcup_{1 \leq s_k^j \leq h_k^j} \theta(t_{s_k^j})(y_k^j) \right)$ to $\theta(t')(v_k)$, where $t_{s_k^j}$ are all the outgoing transitions of δ' at point p_k^j. Note that in the paths leading from p_0 to p_k^j, and then to p, we do not know what is the precise topmost symbol at p_k^j, this is why we consider the union of $\theta\left(t_{s_k^j}\right)(y_k^j)$ over all possible outgoing transitions $t_{s_k^j}$ at p_k^j. This overapproximates the dependence relation.

2 If γ_k^j is of the from $(c_k^j, \top)$, where c_k^j is a constant, then v_k depends on the constant value $(c_k^j, \top)$. Thus, we add $\bigcup_{\gamma_k^j = (c_k^j, \top)} \{(c_k^j, \top)\}$ to $\theta(t')(v_k)$.

Item $\beta_{3.1}$ expresses that, in p', the values of the global variables that are not the function arguments are the same as in p.

Item $\beta_{3.2}$ expresses that the values of the local variables are the same as in p after the call: we record in t'' the dependence relation of the local variables of the caller, so

that we can restore them when the function f returns: when $\langle x_f, \gamma\rangle\ \langle addr, \epsilon\rangle \in \Delta$ is a rule corresponding to the return instruction ret of the function f, item $\beta_{4.1}$ recovers the dependence relation of the local variables after the call [these will be taken from the above $\theta(t'')$ since transitions t^1 in item β_4 correspond to transitions t'' in item β_3: they correspond to return points of the function call]; whereas item $\beta_{4.0}$ updates the dependence relation of the global variables after the call (they are the same as in x_f, i.e., as for the transition t^0).

Item β_5 expresses that if $\langle p, \gamma\rangle\ \langle p', \epsilon\rangle \in \Delta$ is a rule corresponding to a $pop\ y$ instruction, then the values of all the variables that are different from y in p' are the same as in p (item $\beta_{5.0}$), whereas the value of y at p' depends on the topmost stack symbol γ at p (item $\beta_{5.1}$):

1 If γ is of the form $(c, \top)$, where c is a constant, then y at p' depends on $(c, \top)$.

2 If γ is of the form (p'', y') where $y' \in X$ is a variable and $p'' \in P$ is a control point of the program, the value of y at p' depends on the value of y' at p'' [remember, the stack symbol (p'', y') expresses that y' was pushed into the stack at control point p'']. Thus, we need to determine the dependence of variable y' at control point p''. This dependence is determined by the θ of the transitions that are outgoing from p''. Therefore, item $\beta_{5.1}$ adds $\bigcup_{\gamma_k=(p_k,y_k)} \theta(t_k)(y')$ to $\theta(t'')(y)$, for all the transitions t_k that are outgoing from p''. Note that in the paths leading from p_0 to p'', and then to p, we do not know what is the precise topmost symbol at p'', this is why we consider the union of $\theta(t_k)(y')$ over all possible outgoing transitions t_k at p''. This overapproximates the dependence relation.

Item β_6 deals with the case where $\langle p, \gamma\rangle\ \langle p', \epsilon\rangle$ is a rule corresponding to an instruction of the form $add\ esp, x$: such an instruction does not change the value of any variable.

5.2 The data dependence function **Dep**

Then, we can show that for every location p and every variable x, $\mathbf{Dep}(p, x)$ is overapproximated by the union over all the transitions t_j that are outgoing transitions from p of $\theta(t_j)(x)$.

Theorem 1: Let p be a control point of the program. Let d be the number of transitions that are outgoing from p in the MA $\mathcal{A}'$. Let $t_j = (p, \gamma_j, q_j) \in \delta'$ for every $1 \leq j \leq d$ be such transitions, then for every $x \in \mathbf{X}$

$$\mathbf{Dep}(p, x) \subseteq \bigcup_{1 \leq j \leq d} \theta(t_j)(x).$$

Intuitively, each transition $t_j = (p, \gamma_j, q_j)$ in $\mathcal{A}'$ represents some reachable configurations $\mathcal{C}_j$ at control point p, with γ_j as topmost stack symbol. Since the data dependence function $\theta(t_j)$ associated with this transition represents an overapproximation of the data dependence on the paths leading from the entry point of the program p_0 to these reachable configurations $\mathcal{C}_j$, i.e., for every variable x, $\theta(t_j)(x)$ is an overapproximation of $Dep_\rho(p, x)$ for all paths ρ leading to $\mathcal{C}_j$. Since all the reachable configurations at control point p are represented by all the transitions t_j, $1 \leq j \leq d$, that are outgoing from p, we have that $\mathbf{Dep}(p, x) \subseteq \bigcup_{1 \leq j \leq d} \theta(t_j)(x)$. To

formally prove this theorem, we first need to prove the following lemma:

Lemma 1: Let $\rho = i_1, i_2, \ldots, i_k$ be a path in the program leading to the control point p_k from p_0 such that $p_0 \xrightarrow{i_1} p_1 \xrightarrow{i_2} p_2 \ldots p_{k-1} \xrightarrow{i_k} p_k$. Each instruction i_j in ρ $(1 \leq j \leq k)$ corresponds to a PDS rule of the form $\langle p_{j-1}, \gamma_{j-1}\rangle \hookrightarrow \langle p_j, \omega_j\rangle$. Let $t_j = (p_j, \gamma_j, q_j)$, $1 \leq j \leq k$, be a transition added by the saturation procedure to $\mathcal{A}'$ because of the rule $\langle p_{j-1}, \gamma_{j-1}\rangle \hookrightarrow \langle p_j, \omega_j\rangle$. For every $x \in \mathbf{X}$, we have $Dep_\rho(p_k, x) \subseteq \theta(t_k)(x)$.

Proof: Let $\rho = i_1, i_2, \ldots, i_k$ be a path leading to p_k from p_0. Let $\langle p_{j-1}, \gamma_{j-1}\rangle \hookrightarrow \langle p_j, \omega_j\rangle$, $1 \leq j \leq k$, be a PDS rule corresponding to the instruction i_j. Let t_j = (p_j, γ_j, q_j), $1 \leq j \leq k$, be a transition added to $\mathcal{A}'$ by the saturation procedure because of the rule $\langle p_{j-1}, \gamma_{j-1}\rangle \hookrightarrow \langle p_j, \omega_j\rangle$. Let $x \in \mathbf{X}$ be a variable in the program. We will show that $Dep_\rho(p_k, x) \subseteq \theta(t_k)(x)$ by induction on k.

5.2.1 Base case

$k = 1$, $\rho = i_1$. Initially, $(p_0, \sharp, q_F)$ is the only transition in $\mathcal{A}'$. There are different cases depending on the nature of the instruction i_1.

1 i_1 is an assignment of x to a constant c. By the definition of the data dependence relation in Section 4, we have

$$Dep_\rho(p_1, x) = \{(c, \top)\}. \tag{1}$$

Since the instruction i_1 is an assignment and initially $(p_0, \sharp, q_F)$ is the only transition in $\mathcal{A}'$, $\omega_1 = \sharp$, i.e., the PDS rule corresponding to i_1 is $\langle p_0, \sharp\rangle \hookrightarrow \langle p_1, \sharp\rangle$ by the rules in Subsection 3.3. By the saturation procedure in Section 5, since there is the transition $t_0 = (p_0, \sharp, q_F)$ in $\mathcal{A}'$, the transition $t_1 = (p_1, \sharp, q_F)$ is added to $\mathcal{A}'$. By item $\beta_{1.1.1}$, we get

$$\theta(t_1)(x) := \theta(t_1)(x) \cup \{(c, \top)\}. \tag{2}$$

Thus, from Sections 1 and 2 we get $Dep_\rho(p_1, x) \subseteq \theta(t_1)(x)$.

Moreover, by the definition in Section 4, we have for every $y \in \mathbf{X} \setminus \{x\}$ $Dep_\rho(p_1, y) = Dep_\rho(p_0, y)$. By the definition in Section 4, since p_0 is the entry point of the program, we get $Dep_\rho(p_0, y) = \emptyset$. Therefore, we get $Dep_\rho(p_1, y) \subseteq \theta(t_1)(y)$.

2 i_1 is an assignment of x to an expression $exp(y_1, \ldots y_m)$. By the definition in Section 4, $Dep_\rho(p_k, x) = \{(x, p_{k-1})\} \cup \pi$, where $\pi = \{(y_j, p_\ell) |\ 1 \leq j \leq m$ such that y_j is defined at location p_ℓ, $1 \leq \ell \leq k-1$ and there is no assignment to y_j between p_ℓ and p_{k-1} on the path $\rho\}$. Since $k = 1$, there is not any p_ℓ, $1 \leq j \leq m$ and $1 \leq \ell \leq k-1$, such that y_j is defined at p_ℓ. Hence, $\pi = \emptyset$. Thus, we get

$$Dep_\rho(p_1, x) = \{(x, p_0)\}. \tag{3}$$

Since the instruction i_1 is an assignment and initially $(p_0, \sharp, q_F)$ is the only transition in $\mathcal{A}'$, $\omega_1 = \sharp$, i.e., the PDS rule corresponding to i_1 is $\langle p_0, \sharp\rangle \hookrightarrow \langle p_1, \sharp\rangle$

by the rules in Section 3.3. By the saturation procedure in Subsection 5, since there is the transition $t_0 = (p_0, \sharp, q_F)$ in $\mathcal{A}'$, the transition $t_1 = (p_1, \sharp, q_F)$ is added to $\mathcal{A}'$. By item $\beta_{1.1.2}$, we get

$$\theta(t_1)(x) := \theta(t_1)(x) \cup \{(x, p_0)\} \cup \left(\bigcup_{y' \in \text{Var}(exp)} \theta(t_0)(y') \right). \tag{4}$$

Thus, from equations (3) and (4) we get $Dep_\rho(p_1, x) \subseteq \theta(t_1)(x)$.

Moreover, by the definition in Section 4, for any $y \in \mathbf{X} \setminus \{x\}$, $Dep_\rho(p_1, y) = Dep_\rho(p_0, y)$. Since by the definition in Section 4 $Dep_\rho(p_0, y) = \emptyset$, $Dep_\rho(p_1, y) = \emptyset$. Therefore, we get $Dep_\rho(p_1, y) \subseteq \theta(t_1)(y)$.

3 i_1 is an instruction of the form $pop\ x$. This case is not possible since initially the PDS stack is empty.

4 i_1 is a call statement to the function $f(v_1 \dots v_m)$. Let e_f and x_f be the entry point and the exit point of the function f, respectively. Since on the path ρ there is only an instruction i_1 and since initially the PDS stack is empty (contains only the special symbol $\sharp$), f cannot retrieve the values of its parameter from the PDS stack. Thus, in this case, necessarily f has no parameter.

Since initially, $(p_0, \sharp, q_F)$ is the only transition in $\mathcal{A}'$ and i_1 is a call statement, necessarily $\omega_1 = \gamma_{e_f}\sharp$, i.e., the PDS rule corresponding to i_1 is $\langle p_0, \sharp\rangle \hookrightarrow \langle e_f, \gamma_{e_f}\sharp\rangle$ by the rules in Subsection 3.3. By the saturation procedure, since the transition $t_0 = (p_0, \sharp, q_F)$ is in $\mathcal{A}'$, we add $t_1 = (e_f, \gamma_{e_f}, q_{e_f})$ and $t'_1 = (q_{e_f}, \sharp, q_F)$ to $\mathcal{A}'$. By the definition in Section 4, for every $y \in \mathbf{X}$, we have $Dep_\rho(p_1, y) = Dep_\rho(p_0, y)$. Since p_0 is the entry point, we get $Dep_\rho(p_0, y) = \emptyset$ by the definition in Section 4. Hence, $Dep_\rho(p_1, y) = \emptyset$. Therefore, we get $Dep_\rho(p_1, y) \subseteq \theta(t_1)(y)$ and $Dep_\rho(p_1, y) \subseteq \theta(t'_1)(y)$.

5 i_1 is a return statement. This case is not possible since initially the PDS stack is empty.

6 i_1 is a push instruction. By the definition in Section 4, for every $x \in \mathbf{X}$, $Dep_\rho(p_1, x) = Dep_\rho(p_0, x)$. Since p_0 is the entry point, $Dep_\rho(p_0, x) = \emptyset$. Hence, we have $Dep_\rho(p_1, x) = \emptyset$. Since the instruction i_1 is of the form $push\ x$ and initially $(p_0, \sharp, q_F)$ is the only transition in $\mathcal{A}'$, $\omega_1 = \gamma_1\sharp$, i.e., the PDS rule corresponding to i_1 is $\langle p_0, \sharp\rangle \hookrightarrow \langle p_1, \gamma_1\sharp\rangle$. By the saturation procedure in Section 5, since $t_0 = (p_0, \sharp, q_F)$ is in $\mathcal{A}'$, the transitions $t_1 = (p_1, \gamma_1, q')$ and $t'_1 = (q', \sharp, q_F)$ are added to $\mathcal{A}'$. Since $Dep_\rho(p_1, x) = \emptyset$, we get $Dep_\rho(p_1, x) \subseteq \theta(t_1)(x)$ and $Dep_\rho(p_1, x) \subseteq \theta(t'_1)(x)$ for every $x \in \mathbf{X}$.

7 i_1 is an instruction which does not change the value of any variable in $\mathbf{X}$. By the definition in Section 4, for every $x \in \mathbf{X}$, $Dep_\rho(p_1, x) = Dep_\rho(p_0, x)$. Since p_0 is the entry point, $Dep_\rho(p_0, x) = \emptyset$. Hence, we have $Dep_\rho(p_1, x) = \emptyset$. Since i_1 is an instruction which does not change the value of any variable in $\mathbf{X}$ and initially $(p_0, \sharp, q_F)$ is the only transition in $\mathcal{A}'$, $\omega_1 = \sharp$, i.e., the PDS rule corresponding to i_1 is $\langle p_0, \sharp\rangle \hookrightarrow \langle p_1, \sharp\rangle$. By the saturation procedure in Section 5, for the transition $t_0 = (p_0, \sharp, q_F)$ in $\mathcal{A}'$, the transition $t_1 = (p_1, \sharp, q_F)$ is added to $\mathcal{A}'$. Since $Dep_\rho(p_1, x) = \emptyset$, $Dep_\rho(p_1, x) \subseteq \theta(t_1)(x)$ for every $x \in \mathbf{X}$.

5.2.2 Inductive step

$k > 1$. Let $\rho = i_1, \ldots, i_k$ be a path leading to p_k from p_0. By the induction hypothesis, $Dep_\rho(p_{k-1}, x) \subseteq \theta(t_{k-1})(x)$. We will show that $Dep_\rho(p_k, x) \subseteq \theta(t_k)(x)$. There are different cases depending on the nature of the instruction i_k.

1 i_k is an assignment of x to a constant c. By the definition in Section 4, we get

$$Dep_\rho(p_k, x) = \{(c, \top)\}. \tag{5}$$

Since the instruction i_k is an assignment, $\omega_k = \gamma_{k-1}$, i.e., the PDS rule corresponding to i_k is $\langle p_{k-1}, \gamma_{k-1}\rangle \hookrightarrow \langle p_k, \gamma_{k-1}\rangle$. By the saturation procedure in Section 5, since there is the transition $t_{k-1} = (p_{k-1}, \gamma_{k-1}, q')$ in $\mathcal{A}'$, the transition $t_k = (p_k, \gamma_{k-1}, q')$ is added to $\mathcal{A}'$. By item $\beta_{1.1.1}$, we get

$$\theta(t_k)(x) := \theta(t_k)(x) \cup \{(c, \top)\}. \tag{6}$$

Thus, from equations (5) and (6) we get $Dep_\rho(p_k, x) \subseteq \theta(t_k)(x)$.

Moreover, by the definition in Section 4, for every $y \in \mathbf{X} \setminus \{x\}$, we have

$$Dep_\rho(p_k, y) = Dep_\rho(p_{k-1}, y). \tag{7}$$

By item $\beta_{1.0}$, we get

$$\theta(t_k)(y) := \theta(t_k)(y) \cup \theta(t_{k-1})(y). \tag{8}$$

By the induction hypothesis, we have

$$Dep_\rho(p_{k-1}, y) \subseteq \theta(t_{k-1})(y). \tag{9}$$

Thus, from equations (7), (8) and (9) we get $Dep_\rho(p_k, y) \subseteq \theta(t_k)(y)$.

2 i_k is an assignment of x to an expression $exp(y_1, \ldots y_m)$. Since y_j is defined at p_ℓ, $1 \leq j \leq m$ and $1 \leq \ell \leq k-1$, $(y_j, p_\ell) \in Dep_\rho(p_{k-1}, y_j)$ for $1 \leq j \leq m$. Then, we have

$$\bigcup_{1 \leq j \leq m} (y_j, p_\ell) \subseteq \bigcup_{1 \leq j \leq m} Dep_\rho(p_{k-1}, y_j). \tag{10}$$

By the definition in Section 4, we have

$$Dep_\rho(p_k, x) = \{(x, p_{k-1})\} \cup \bigcup_{1 \leq j \leq m} (y_j, p_\ell) \tag{11}$$

Since the instruction i_k is an assignment, $\omega_k = \gamma_k$, i.e., the PDS rule corresponding to i_k is of the form $\langle p_{k-1}, \gamma_{k-1}\rangle \hookrightarrow \langle p_k, \gamma_k\rangle$. By the saturation procedure in Section 5, since the transition $t_{k-1} = (p_{k-1}, \gamma_{k-1}, q')$ is in $\mathcal{A}'$, the transition $t_k = (p_k, \gamma_{k-1}, q')$ is added to $\mathcal{A}'$. By item $\beta_{1.1.2}$, we get

$$\theta(t_k)(x) := \theta(t_k)(x) \cup \{(x, p_{k-1})\} \cup \left(\bigcup_{1 \leq j \leq m} \theta(t_{k-1})(y_j) \right). \tag{12}$$

By the induction hypothesis, we have

$Dep_\rho(p_{k-1}, y_j) \subseteq \theta(t_{k-1})(y_j)$ for $1 \leq j \leq m$.

Hence, we get

$$\bigcup_{1\leq j\leq m} Dep_\rho(p_{k-1}, y_j) \subseteq \bigcup_{1\leq j\leq m} \theta(t_{k-1})(y_j). \tag{13}$$

Therefore, from equations (11), (10), (12) and (13) we get $Dep_\rho(p_k, x) \subseteq \theta(t_k)(x)$.

Moreover, by the definition in Section 4, for every $y \in \mathbf{X} \setminus \{x\}$, we get

$$Dep_\rho(p_k, y) = Dep_\rho(p_{k-1}, y). \tag{14}$$

By item $\beta_{1.0}$, we get

$$\theta(t_k)(y) := \theta(t_k)(y) \cup \theta(t_{k-1})(y). \tag{15}$$

By the induction hypothesis, we have

$$Dep_\rho(p_{k-1}, y) \subseteq \theta(t_{k-1})(y). \tag{16}$$

Thus, from equations (14), (15) and (16) we get $Dep_\rho(p_k, y) \subseteq \theta(t_k)(y)$.

3 i_k is the instruction of the form $pop\ x$. Since i_k is of the form $pop\ x$, $\omega_k = \epsilon$, i.e., the PDS rule corresponding to i_k is $\langle p_{k-1}, \gamma_{k-1}\rangle \hookrightarrow \langle p_k, \epsilon\rangle$. By the saturation procedure in Section 5, for every transitions $t_{k-1} = (p_{k-1}, \gamma_{k-1}, q)$ and $t'_{k-1} = (q, \gamma', q')$ in $\mathcal{A}'$, the transition $t_k = (p_k, \gamma', q')$ is added to $\mathcal{A}'$. By the definition in Section 4, there are two cases depending on the topmost stack symbol γ':

a If the topmost symbol γ' of the PDS stack corresponding to this execution at p_k is of the form (y, p_ℓ), $1 \leq \ell \leq k-1$, we have

$$Dep_\rho(p_k, x) = Dep_\rho(p_\ell, y). \tag{17}$$

Because i_k is the instruction of the form $pop\ x$. Let $t_j = (p_\ell, \gamma'_j, q'_j)$, $1 \leq j \leq m$, be all transitions in $\mathcal{A}'$ outgoing from p_ℓ [$\gamma'_j = (y'_j, p_\ell)$ is a possible topmost stack symbol at p_ℓ]. Then, by item $\beta_{5.1}$, we have

$$\theta(t_k)(x) := \theta(t_k)(x) \cup \bigcup_{1\leq j\leq m} \left(\bigcup_{\gamma_j=(y_j,p_\ell)} \theta(t_j)(y_j) \cup \bigcup_{\gamma_j=(c_j,\top)} \{(c_j, \top)\} \right).$$

Hence, we get

$$\bigcup_{1\leq j\leq m} \left(\bigcup_{\gamma_j=(y'_j,p_\ell)} \theta(t_j)(y'_j) \right) \subseteq \theta(t_k)(x). \tag{18}$$

And there exists a transition t_j at p_ℓ such that γ'_j correspond to γ' and y'_j corresponds to y in equation (17).

By the induction hypothesis, we have

$$Dep_\rho(p_\ell, y) \subseteq \theta(t_j)(y'_j) \text{ where } y = y'_j. \tag{19}$$

Therefore, from equations (17), (18) and (19) we get $Dep_\rho(p_k, x) \subseteq \theta(t_k)(x)$.

b If the topmost symbol of the PDS stack corresponding to this execution is $(c, \top)$, by the definition in Section 4, we have

$$Dep_\rho(p_k, x) = \{(c, \top)\}. \tag{20}$$

By item $\beta_{5.1}$, we have

$$\theta(t_k)(x) := \theta(t_k)(x) \cup \{(c, \top)\}. \tag{21}$$

Thus, from equations (20) and (21) we get $Dep_\rho(p_k, x) \subseteq \theta(t_k)(x)$.

Moreover, by the definition in Section 4, for every $y \in \mathbf{X} \setminus \{x\}$, we have

$$Dep_\rho(p_k, y) = Dep_\rho(p_{k-1}, y). \tag{22}$$

By item $\beta_{5.0}$, we have

$$\theta(t_k)(y) := \theta(t_k)(y) \cup \theta(t_{k-1})(y). \tag{23}$$

By the induction hypothesis, we have

$$Dep_\rho(p_{k-1}, y) \subseteq \theta(t_{k-1})(y). \tag{24}$$

Thus, from equations (22), (23) and (24) we get $Dep_\rho(p_k, y) \subseteq \theta(t_k)(y)$.

4 i_k is a call statement to the function $f(v_1 \dots v_m)$. Then, p_k is the return address of the function f. Let e_f and x_f be the entry point and the exit point of the function f, respectively. We will consider the data dependence function at e_f and x_f as follows.

Let us consider the variables at the entry point e_f of the function. Since i_k is of the form $call\ f(v_1 \dots v_m)$, $\omega_k = \gamma_{e_f}\gamma_{k-1}$, i.e., the PDS rule corresponding to i_k is $\langle p_{k-1}, \gamma_{k-1}\rangle \hookrightarrow \langle e_f, \gamma_{e_f}\gamma_{k-1}\rangle$. By the saturation procedure in Section 5, for the transition $t_{k-1} = (p_{k-1}, \gamma_{k-1}, q)$ in $\mathcal{A}'$, the transitions $t_{e_f} = (e_f, \gamma_{e_f}, q_{e_f})$ and $t'_{e_f} = (q_{e_f}, \gamma_{k-1}, q)$ are added to $\mathcal{A}'$. Let d be the number of different prefixes of length m of accepting paths in the MA $\mathcal{A}'$ starting from the transition t_{k-1}. Let $\gamma_1^j \cdots \gamma_m^j$, for $j, 1 \le j \le d$, be such prefixes ($\gamma_1^j = \gamma_{k-1}$ for every $j, 1 \le j \le d$). Hence, the h^{th} element on top of the stack belongs to $\{\gamma_h^j | 1 \le j \le d\}$ for every h, $1 \le h \le m$. Then, γ_h^j is either of the form (y_h^j, p_h^j), where $y_h^j \in X$ is a variable and $p_h^j \in P$ is a control point of the program; or of the form $(c_h^j, \top)$, where c_h^j is a constant. If γ_h^j is of the form (p_h^j, y_h^j), $1 \le h \le m$, $1 \le j \le d$, let $t_{s_h^j}$,

$1 \leq s_h^j \leq \ell_h^j$ be all transitions in $\mathcal{A}'$ of the form $t_{s_h^j} = (p_h^j, \gamma'_{s_h^j}, q_{s_h^j})$. By item $\beta_{3.0}$, for every parameter v_h ($1 \leq h \leq m$), we get

$$\theta(t_{e_f})(v_h) := \theta(t_{e_f})(v_h) \bigcup_{1 \leq j \leq d} \left(\bigcup_{\gamma_h^j = (y_h^j, p_h^j)} \left(\bigcup_{1 \leq s_h^j \leq l_h^j} \theta(t_{s_h^j})(y_h^j) \right) \cup \bigcup_{\gamma_h^j = (c_h^j, \top)} \{(c_h^j, \top)\} \right) \tag{25}$$

By the definition in Section 4, for every parameter v_h ($1 \leq h \leq m$) at the entry point of the function f, there are two cases:

a If the h^{th} element on top of the stack is of the form (y_h, p_h), we have

$$Dep_\rho(e_f, v_h) = Dep_\rho(p_h, y_h). \tag{26}$$

We have $(y_h, p_h) \in \{\gamma_h^j | 1 \leq j \leq d\}$ since $\{\gamma_h^j | 1 \leq j \leq d\}$ is a set of the h^{th} symbols in different prefixes of the stack with length m. Hence, there exists a transition $t_{s_h^j} \in \mathcal{A}'$ corresponding to stack symbol $(y_h, p_h) \in \{\gamma_h^j | 1 \leq j \leq d\}$. By the induction hypothesis, we have

$$Dep_\rho(p_h, y_h) \subseteq \theta(t_{s_h^j})(y_h). \tag{27}$$

Thus, from equations (25), (27) and (26) we get $Dep_\rho(e_f, v_h) \subseteq \theta(t_{e_f})(v_h)$.

b If the h^{th} element on top the stack is of the form $(c_h, \top)$, we have

$$Dep_\rho(e_f, v_h) = \{(c_h, \top)\}. \tag{28}$$

We have $(c_h, \top) \in \{\gamma_h^j | 1 \leq j \leq d\}$ since $\{\gamma_h^j | 1 \leq j \leq d\}$ is a set of the h^{th} symbol in different prefixes of the stack with length m. Hence, there exists γ_h^j such that $\gamma_h^j = (c_h, \top)$ such that

$$\{(c_h, \top)\} \subseteq \bigcup_{\gamma_h^j = (c_h^j, \top)} \{(c_h^j, \top)\}. \tag{29}$$

Thus, from equations (25), (28) and (29) we get $Dep_\rho(e_f, v_h) \subseteq \theta(t_{e_f})(v_h)$.

Moreover, by the definition in Section 4, for every $y' \in \mathbf{X}_{global} \setminus \{v_h | 1 \leq h \leq m\}$, we have

$$Dep_\rho(e_f, y') = Dep_\rho(p_{k-1}, y'). \tag{30}$$

By item $\beta_{3.1}$, every $y' \in \mathbf{X}_{global} \setminus \{v_h | 1 \leq h \leq m\}$, we get

$$\theta(t_{e_f})(y') := \theta(t_{e_f})(y') \cup \theta(t_{k-1})(y'). \tag{31}$$

By the induction hypothesis, we have

$$Dep_\rho(p_{k-1}, y') \subseteq \theta(t_{k-1})(y'). \tag{32}$$

Therefore, from equations (30), (31) and (32) we get
$Dep_\rho(e_f, y') \subseteq \theta(t_{e_f})(y')$ for every $y' \in \mathbf{X}_{global} \setminus \{v_h | 1 \leq h \leq m\}$.

Besides, by the definition in Section 4, for every
$y' \in \mathbf{X}_{local} \setminus \{v_h | 1 \leq h \leq m\}$, we have

$$Dep_\rho(e_f, y') = Dep_\rho(p_{k-1}, y'). \tag{33}$$

By item $\beta_{3.2}$, every $y' \in \mathbf{X}_{local} \setminus \{v_h | 1 \leq h \leq m\}$, we get

$$\theta(t'_{e_f})(y') := \theta(t'_{e_f})(y') \cup \theta(t_{k-1})(y'). \tag{34}$$

By the induction hypothesis, we have

$$Dep_\rho(p_{k-1}, y') \subseteq \theta(t_{k-1})(y'). \tag{35}$$

Therefore, from equations (33), (34) and (36) we get
$Dep_\rho(e_f, y') \subseteq \theta(t'_{e_f})(y')$ for every $y' \in \mathbf{X}_{local} \setminus \{v_h | 1 \leq h \leq m\}$.

5 As for the return statement, i.e., ret, at x_f of the function f, let us consider the variables at the exit point x_f of the function. At the exit point x_f of the function $f(v_1 \ldots v_m)$, there exists a return statement corresponding to the PDS rule $\langle x_f, \gamma_{x_f} \rangle \hookrightarrow \langle p_k, \epsilon \rangle \in \Delta$ where γ_{x_f} is of the form $(p_k, \top)$.

By the saturation procedure in Section 5, since transitions $t_{x_f} = (x_f, \gamma_{x_f}, q_{e_f})$ and $t'_{e_f} = (q_{e_f}, \gamma_{k-1}, q)$ are in $\mathcal{A}'$, the transition $t_k = (p_k, \gamma_{k-1}, q)$ is added to $\mathcal{A}'$. Note that $t'_{e_f} = (q_{e_f}, \gamma_{k-1}, q)$ is added to $\mathcal{A}'$ since i_k is a call statement and the transition $t_{k-1} = (p_{k-1}, \gamma_{k-1}, q)$ is in $\mathcal{A}'$ (see item 4).

By the definition in Section 4, for every global variable $y \in \mathbf{X}_{global}$, we have

$$Dep_\rho(p_k, y) = Dep_\rho(x_f, y). \tag{36}$$

By the induction hypothesis, we have

$$Dep_\rho(x_f, y) \subseteq \theta(t_{x_f})(y). \tag{37}$$

By item $\beta_{4.0}$, for every global variable $y \in \mathbf{X}_{global}$, we get

$$\theta(t_k)(y) := \theta(t_k)(y) \cup \theta(t_{x_f})(y). \tag{38}$$

Therefore, from equations (36), (37) and (38) we get $Dep_\rho(p_k, y) \subseteq \theta(t_k)(y)$.

Moreover, by item $\beta_{3.2}$, for every $y \in \mathbf{X}_{local}$ we get

$$\theta(t'_{e_f})(y) := \theta(t'_{e_f})(y) \cup \theta(t_{k-1})(y). \tag{39}$$

By item $\beta_{4.1}$, we get

$$\theta(t_k)(y) := \theta(t_k)(y) \cup \theta(t'_{e_f})(y). \tag{40}$$

Thus, from equations (39) and (40) we get

$$\theta(t_{k-1})(y) \subseteq \theta(t_k)(y). \tag{41}$$

By the definition in Section 4, for every $y \in \mathbf{X}_{local}$, we have

$$Dep_\rho(p_k, y) = Dep_\rho(p_{k-1}, y). \tag{42}$$

By the induction hypothesis, we have

$$Dep_\rho(p_{k-1}, y) \subseteq \theta(t_{k-1})(y). \tag{43}$$

Therefore, from equations (41), (42) and (43) we get $Dep_\rho(p_k, y) \subseteq \theta(t_k)(y)$.

6 i_k is a push instruction. By the definition in Section 4, we have

$$Dep_\rho(p_k, y) = Dep_\rho(p_{k-1}, y). \tag{44}$$

Since the instruction i_k is of the form $push\ y$, $\omega_k = \gamma_k\gamma_{k-1}$, i.e., the PDS rule corresponding to i_k is $\langle p_{k-1}, \gamma_{k-1}\rangle \hookrightarrow \langle p_k, \gamma_k\gamma_{k-1}\rangle$. By the saturation rule in Section 5, for every transition $t_{k-1} = (p_{k-1},\ \gamma_{k-1}, q) \in \mathcal{A}'$, we add $t_k = (p_k, \gamma_k, q')$ and $t'_k = (q', \gamma_{k-1}, q)$ to $\mathcal{A}'$. By item $\beta_{2.0}$, we have

$$\theta(t_k)(y) := \theta(t_k)(y) \cup \theta(t_{k-1})(y). \tag{45}$$

By item $\beta_{2.1}$, we have

$$\theta(t'_k)(y) := \theta(t'_k)(y) \cup \theta(t_{k-1})(y). \tag{46}$$

By the induction hypothesis, we have

$$Dep_\rho(p_{k-1}, y) \subseteq \theta(t_{k-1})(y). \tag{47}$$

Therefore, from equations (44), (45), (46) and (47) we get $Dep_\rho(p_k, y) \subseteq \theta(t_k)(y)$ and $Dep_\rho(p_k, y) \subseteq \theta(t'_k)(y)$.

7 i_k is an instruction which does not change the value of any variable in **X**. By the definition in Section 4, we have

$$Dep_\rho(p_k, y) = Dep_\rho(p_{k-1}, y). \tag{48}$$

Since i_k is an instruction which does not change the value of any variable in **X**, $\omega_k = \gamma_{k-1}$, i.e., the PDS rule corresponding to i_k is $\langle p_{k-1}, \gamma_{k-1}\rangle \hookrightarrow \langle p_k, \gamma_{k-1}\rangle$. By the saturation procedure in Section 5 , for every transition $t_{k-1} = (p_{k-1}, \gamma_{k-1}, q) \in \mathcal{A}'$, the transition $t_k = (p_k, \gamma_{k-1}, q)$ is added to $\mathcal{A}'$. By item $\beta_{1.0}$, we get

$$\theta(t_k)(y) := \theta(t_k)(y) \cup \theta(t_{k-1})(y). \tag{49}$$

By the induction hypothesis, we have

$$Dep_\rho(p_{k-1}, y) \subseteq \theta(t_{k-1})(y). \tag{50}$$

Thus, from equations (48), (49) and (50) we get $Dep_\rho(p_k, y) \subseteq \theta(t_k)(y)$.

□

We are now ready to prove Theorem 1:

Proof of Theorem 1: Let $x \in \mathbf{X}$ be a variable. Let $\rho_j = i_1^j \cdots i_{k_j}^j$, $1 \le j \le s$, be all the paths leading to p from the entry point p_0 of the program. For every instruction $i_{h_j}^j \in \rho_j$, $1 \le h_j \le k_j$, there exists a transition $t_j = (p, \gamma_j, q_j)$ that is added to $\mathcal{A}'$. By the definition at the end of Section 4,

$$\mathbf{Dep}(p, x) = \bigcup_{1 \le j \le s} Dep_{\rho_j}(p, x).$$

By Lemma 1, $Dep_{\rho_j}(p, x) \subseteq \theta(t_j)(x)$. Hence, we get $\bigcup_{1\le j\le s} Dep_{\rho_j}(p, x) \subseteq \bigcup_{1\le j\le s} \theta(t_j)(x)$.

Thus, $\mathbf{Dep}(p, x) \subseteq \bigcup_{1\le j\le s} \theta(t_j)(x)$. □

6 The extended API call graph

We show in this section how to compute the extended API call graph of a given program using the data dependence function computed in Section 5.

6.1 Definition

Let $\mathcal{F}$ be the set of all API functions that are called in the program. For every API function $f \in \mathcal{F}$, let $Para(f)$ be the set of parameters of f and $|Para(f)|$ be the number of parameters of f. For each API function, there can be special parameters on which the behaviour (the output) of the API function depends, e.g., calling the API function `GetModuleFileName` with 0 as first parameter returns the current file path of this execution, thus the value of the first parameter is crucial for the nature of the output of

the call to `GetModuleFileName`. We call such parameters *meaningful* parameters. Let $Para_M(f)$ be such *meaningful* parameters of f.

An extended API call graph is a directed graph $\mathcal{G} = (V, E)$ such that: $V = V_1 \cup V_2$, where $V_1 \subseteq \{(f, eval) \mid f \in \mathcal{F}, Para_M(f) \neq \emptyset,$ and $eval : Para_M(f) \rightarrow 2^{\mathbb{Z}}$ is a function $\}$ is the set of vertices consisting of pairs of the form $(f, eval)$ for an API function f and an evaluation $eval$ that specifies the value of the meaningful parameters of f, and $V_2 \subseteq \{f \mid f \in \mathcal{F}, Para_M(f) = \emptyset\}$ is the set of vertices labelled by API functions with no meaningful parameter.

Let $v \in V$. Let $f \in \mathcal{F}$ be such that v of the form $(f, eval)$ or f. Then, we define $func(v) = f$ and $Para(v) = Para(f)$. Moreover, if v of the form $(f, eval)$, then $mean(v) = eval$, and if v of the form f, then $mean(v) = \emptyset$.

$E \subseteq \{(v_1, 2^{|Para(v_1)| \times |Para(v_2)|}, v_2) \mid v_1$ and $v_2 \in V\}$ is the set of edges. $(v_1, e, v_2) \in E$ means that the API function $func(v_1)$ with meaningful parameter values defined by $mean(v_1)$ is called before the API function $func(v_2)$ with meaningful parameter values defined by $mean(v_2)$. Moreover, $(i, j) \in e$ means that the i^{th} parameter of $func(v_1)$ and the j^{th} parameter of $func(v_2)$ are related.

In the rest of the paper, we can abuse terminology as follows: if $v = (f, eval)$ is a vertex, we can say that f is a node of the graph labelled with $eval$. Moreover, if (v_1, e, v_2) is an edge, where $e \in 2^{|Para(v_1)| \times |Para(v_2)|}$, we can say that $(func(v_1), func(v_2))$ is an edge labelled by e.

For example, let us consider the extended API call graph of Figure 1(c). Let a_1 represent the first argument of function GetModuleFileName. Here, a_1 is meaningful, whereas CopyFile does not have any meaningful parameter. Thus, $V = \{(GetModuleFileName, eval), \ CopyFile \mid eval(a_1) = \{0\}\}$: $eval(a_1) = \{0\}$ expresses that when GetModuleFileName is called, the first parameter has to be equal to 0. $E = \{((GetModuleFileName, eval), (2, 1), CopyFile)\}$: the pair (2, 1) expresses that the second parameter of GetModuleFileName serves as first parameter of CopyFile.

Graphical representation: Note that in our graphical representation in Figure 1(c), $\bar{1} = \{0\}$ represents $eval(a_1) = \{0\}$, whereas $\bar{2} \rightsquigarrow \bar{1}$ stands for (2, 1).

6.2 Computing the extended API call graph

Let $\mathcal{A} = (Q, \Gamma, \delta, Q_0, Q_F)$ be a MA of the PDS $\mathcal{P} = (P, \Gamma, \Delta)$, where $Q = P \cup \{q_F\}$, $Q_0 = \{p_0\}$ (p_0 is the entry point of the program), $Q_F = \{q_F\}$ and $\delta = \{(p_0, \sharp, q_F)\}$. $\mathcal{A}$ accepts the initial configuration of the PDS $\langle p_0, \sharp \rangle$. Let $\mathcal{A}' = (Q', \Gamma, \delta', Q_0, Q_F)$ be the MA that accepts $post^*(\langle p_0, \sharp \rangle)$ as described in Section 5, and let θ be the annotation function as computed in Section 5.

Let $P_{API} \subset P$ be the set of control points where there are calls to API functions. For each $p \in P_{API}$, let $\eta(p)$ (resp. $\iota(p)$) be the name (resp. the number of parameters) of the API function called at point p. Let $\mathcal{A}^\psi$ be the MA obtained from $\mathcal{A}'$ by considering only API function calls and their corresponding parameters (i.e., we keep only control points in P_{API} corresponding to API function calls, and we cut the stack of $\mathcal{A}'$ in order to keep only the part of the stack that contains the parameters of these function calls). More precisely, let $\mathcal{A}^\psi$ be the MA that accepts $\{\langle p, \omega \rangle \in P_{API} \times \Gamma^* \mid |\omega| = \iota(p)$ and $\exists \omega' \in \Gamma^* s.t. \langle p, \omega\omega' \rangle \in L(\mathcal{A}')\}$. $\mathcal{A}^\psi$ can be easily computed from $\mathcal{A}'$. Let ψ be the operator that performs this operation: $\psi(\mathcal{A}') = \mathcal{A}^\psi$. Let $\langle p, \omega \rangle \in \mathcal{A}^\psi$. Let $\mathcal{A}_p^\omega$ be an MA that accepts the set of configurations $\{\langle p, \omega\omega' \rangle \in L(\mathcal{A}')\}$, i.e., $\mathcal{A}_p^\omega$ accepts the set of reachable configurations that are in control point p and that have ω as parameter

values of the API $\eta(p)$. For each configuration $c = \langle p, \omega \rangle \in \mathcal{A}^{\psi}$, let $f = \eta(p)$ be the API function called at p. Let m be the number of parameters of f. Then c is necessarily of the form $c = \langle p, \gamma_1 \cdots \gamma_m \rangle$. We define $mean(c)$, the function that evaluates the values of the meaningful parameters of the API function $\eta(p)$ at configuration c as follows:

- If f has no meaningful parameter (i.e., $Para_M(f) = \emptyset$), $mean(c) = \emptyset$.
- Otherwise, let $Para_M(f) = \{a_{i_1}, \ldots, a_{i_k}\}$ be the meaningful parameters of f, $1 \leq i_j \leq m$, $1 \leq j \leq k$:

  1 If γ_{i_j} is of the form $(c, \top)$, then $mean(c)(a_{i_j}) = \{c\}$.

  2 If γ_{i_j} is of the form (x_{i_j}, p_{i_j}), where x_{i_j} is a variable in X and p_{i_j} is a control point (that is necessarily in $\mathcal{A}'$ since $\mathcal{A}'$ represents all the reachable configurations), then $mean(c)(a_{i_j}) = \mathcal{O}(p_{i_j})(x_{i_j})$. [Remember, (x_{i_j}, p_{i_j}) is in the stack because x_{i_j} is pushed at control point p_{i_j}. Thus, we get from the oracle the value of x_{i_j} at control point p_{i_j}].

We compute the extended API call graph in two phases as follows. The first phase computes the nodes of the graph and their labels by using the oracle $\mathcal{O}$ (remember that, as mentionned in Section 3, we use an oracle to get the possible values of a variable at each program's location: $\mathcal{O}(p)(x)$ gives all possible values of variable x at the program location p. In our implementation, we use Jakstab (Kinder and Veith, 2008) and IDA Pro (Eagle, 2011) to get this oracle). The second phase computes the links between nodes in the graph by using the data dependence function **Dep**. Intuitively, the first phase computes the set of nodes from the API functions that are called at each location in the program. For each configuration $\langle p, \omega \rangle \in L(\mathcal{A}^{\psi})$, the API function called at p, together with the value of its meaningful parameters at $\langle p, \omega \rangle$ are added to the graph. This API function name is given by the operator $\eta(p)$. The values of the meaningful parameters of each API function are determined by the stack content. The edges are computed in the second phase: Given two nodes v_1 and v_2 corresponding to configurations $\langle p, \omega \rangle \in L(\mathcal{A}^{\psi})$ and $\langle p', \omega' \rangle \in L(\mathcal{A}^{\psi})$, respectively, we add an edge from v_1 to v_2 if $\langle p', \omega' \rangle$ can be reached from $\langle p, \omega \rangle$. The data dependence function **Dep** is used to add labels to the edges of the graph that describe the data dependence relation between the parameters of the different API functions.

More precisely, the extended API call graph $G = (V, E)$ is computed as follows:

1 Initially, $V = \emptyset$ and $E = \emptyset$.

2 We compute the set of nodes V as follows: For each configuration $c = \langle p, w \rangle$ in $L(\psi(\mathcal{A}'))$, let $f = \eta(p)$ be the API function called at p. We add to V the node $(f, mean(c))$, where if f has no meaningful parameter and $mean(c) = \emptyset$, then $\big(f, mean(c)\big)$ stands for the node f.

3 We compute the set of edges E as follows: for every configuration $c = \langle p, \omega \rangle \in L(\psi(\mathcal{A}'))$, for every configuration $c' = \langle p', \omega' \rangle \in L\big(\psi(post^*(L(\mathcal{A}_p^{\omega}))\big)$ [starting from control p, with ω as values for the parameters of $\eta(p)$, we can reach control point p' with ω' as values for the parameters of $\eta(p')$]. Let $f = \eta(p)$ and $f' = \eta(p')$ be the API functions called at p and p', respectively. Let m (resp. m') be the number of parameters of f (resp.

f'). Then, necessarily, ω is of the form $\gamma_1 \cdots \gamma_m$ and ω' is of the form $\gamma'_1 \cdots \gamma'_{m'}$. Let $e = \emptyset$. For every $1 \leq h \leq m$ and $1 \leq k \leq m'$:

- If γ_h is of the form (x_h, p_h), where x_h is a variable and p_h is a control point of the program, then:
  - a If γ'_k is of the form (x'_k, p'_k), where x'_k is a variable and p'_k is a control point of the program, then if $\mathbf{Dep}(p, x_h) \cap \mathbf{Dep}(p', x'_k) \neq \emptyset$, we add (h, k) to e.
  - b If γ'_k is of the form $(c_k, \top)$, for a constant c_k, then if $(c_k, \top) \in \mathbf{Dep}(p, x_h)$, we add (h, k) to e.
- If γ_h is of the form $(c_h, \top)$ for a constant c_h, then:
  - a If γ'_k is of the form (x'_k, p'_k), where x'_k is a variable and p'_k is a control point of the program, then if $(c_h, \top) \in \mathbf{Dep}(p', x'_k)$, we add (h, k) to e.
  - b If γ'_k is of the form $(c_k, \top)$, then if $c_k = c_h$, we add (h, k) to e.

Then, we add $\Big(\big(f, mean(c)\big), e, \big(f', mean(c')\big)\Big)$ to E. Note that from the above construction, (h, k) is added to e means that the h^{th} parameter of f and the k^{th} parameter of f' are related (they depend on the same variable or on the same value). Thus, we add (h, k) to the set labelling the edge corresponding to these configurations c and c' and relating f to f'.

7 Extracting malicious behaviours

Given a set of extended API call graphs that correspond to malwares and a set of extended API call graphs corresponding to benign programs, we want to extract in a *completely automatic way* a malicious API graph that corresponds to the malicious behaviours of the malwares. This malicious extended API graph should represent the parts of the API call graphs of the malwares that correspond to the malicious behaviours. Thus, we need to retrieve from these graphs the set of subgraphs that are relevant for malicious behaviours, and discard the nonrelevant (benign) ones. As mentionned in the introduction, this can be seen as an IR problem. The IR community has proposed several techniques that have been shown to be efficient for text and image retrieval. Our goal is to adapt these techniques to extract malicious API graphs. We show in this section how this can be done. In what follows, we first recall the techniques that are used in the IR community, and then we show how they can be adapted to our problem.

7.1 Term weighting scheme

IR consists of retrieving documents with relevant information from a given set of documents (a collection). Web search, email search, etc. are IR examples. Effective retrieval ensures that items likely to be relevant are retrieved, whereas items likely to be irrelevant must be rejected. This problem was extensively studied the last 30 years. IR research has focused on the retrieval of text documents and images. The IR

community came up with several techniques that were proven to be efficient. All these techniques are based on extracting from each document the set of terms that allow to distinguish this document from the other documents in the collection. This is achieved by associating a weight to each term in every document in the collection. The term weight represents the relevance of a term in a document. The higher the term weight is, the more relevant the term is in the document. A large number of weighting functions have been investigated. An intuitive and well known scheme of term weighting that has proven itself to be efficient is the TFIDF scheme, where TF stands for term frequency and IDF stands for inverse document frequency. The TFIDF term weight is measured from the occurrences of terms in a document and their appearances in other documents. A term is relevant to a document if it occurs frequently in this document and rarely appears in other documents. Intuitively, a common term which appears in a lot of documents is not relevant (like 'the', 'a', 'with', 'of', etc. are terms that can be found in every document but are irrelevant) and a term which occurs frequently in a document and rarely in the other documents is more relevant to this document. According to this scheme, the weight of a term i in document j is formally defined as follows.

$$w(i,j) = \mathbf{tf}(i,j) \times \mathbf{idf}(i) \tag{51}$$

where $\mathbf{tf}(i,j)$ is the number of occurrences of term i in document j, called term frequency. The **idf** factor ensures that terms concentrated in a few documents of a collection are favoured. It varies inversely with the number of documents $\mathbf{df}(i)$ to which a term i is assigned in a collection of N documents. A typical factor may be computed as $\log(\frac{N}{\mathbf{df}(i)})$.

To improve the performances of this weighting scheme, several adjustements can be made:

- The weights in the document can be normalised by the length of the document, so that long documents are not automatically favoured over short documents. Indeed, term frequencies are usually bigger for longer documents. A normalisation factor can then be introduced (Singhal et al., 1996). The length normalisation component can be computed as follows:

  $$\frac{\mathbf{S}(j)}{\mathbf{AVG}(\mathcal{D})} \times b + (1-b), 0 \leq b \leq 1$$

  where $\mathbf{S}(j)$ is the length of document j and $\mathbf{AVG}(\mathcal{D})$ is the average length of documents in the collection $\mathcal{D}$. In the above formula, by setting b to 1, document length normalisation is fully performed, while setting b to 0 turns off the normalisation effect.

- To ensure that high **tf** for a relevant term in a document does not place that document ahead of other documents which have multiple relevant terms but with lower **tf** values, the logarithmic **tf**-factor $(1 + ln\big(1 + ln(\mathbf{tf})\big))$ by referring to Singhal et al. (1999) and Singhal and Kaszkiel (2001) or the BM25 **tf**-factor $(\mathbf{tf}/(k + \mathbf{tf})$ for some $k > 0)$ by referring to Robertson and Zaragoza (2009) and Robertson et al. (1995) or the sigmoid **tf**-factor $(1/(1 + e^{-\mathbf{tf}}))$ by referring to Yao et al. (2006) can be introduced, since these functions increase slowly wrt **tf**, not like raw **tf**.

Thus, to obtain better performances, the **tf** factor can be replaced by $F(\mathbf{tf})$, for a function F that can be defined as follows:

- $F_1(\mathbf{tf}(i,j)) = \mathbf{tf}(i,j)$ leads to the raw factor.
- $F_2(\mathbf{tf}(i,j)) = \frac{(k_1+1)\times \mathbf{tf}(i,j)}{\mathbf{tf}(i,j)+k_1(\frac{\mathbf{S}(j)}{\mathbf{AVG}(\mathcal{D})}\times b+1-b)}$ implements the length normalised BM25 factor.
- The length normalised logarithmic factor can be implemented by function F_3 defined as follows:

$$\begin{cases} \frac{1+ln(1+ln(\mathbf{tf}(i,j)))}{\frac{\mathbf{S}(j)}{\mathbf{AVG}(\mathcal{D})}\times b+1-b} & \text{if } \mathbf{tf}(i,j) > 0 \\ 0 & \text{if } \mathbf{tf}(i,j) = 0 \end{cases}$$

- The length normalised sigmoid factor can be implemented by function F_4 defined as follows:

$$\begin{cases} \frac{k_1+1}{k_1(\frac{\mathbf{S}(j)}{\mathbf{AVG}(\mathcal{D})}\times b+1-b)+e^{-\mathbf{tf}(i,j)}} & \text{if } \mathbf{tf}(i,j) > 0 \\ 0 & \text{if } \mathbf{tf}(i,j) = 0 \end{cases}$$

All these functions were applied in IR and have shown good performances. There is no real theoretical reason why these functions are used. They are used just because they work well. Depending on the application, one function can be better than the others.

7.2 Term weights in extended API call graphs

An extended API call graph is a set of nodes and edges. Thus, to extract the relevant subgraphs, it is sufficient to extract the relevant nodes and edges that compose it. Our goal is then to isolate the few relevant nodes and edges from the nonrelevant ones. To this aim, we follow the IR community and associate a TFIDF weight to each node and each edge in the extended API call graphs of the collection.

Let then $\mathcal{G}$ be a set of extended API call graphs. Let i be a term (either a node or an edge), and j be an extended API call graph. Then,

$$w_g(i,j) = F\big(\mathbf{tf}(i,j)\big) \times \mathbf{idf}(i) \tag{52}$$

here $\mathbf{tf}(i,j)$ is the number of occurrences of the term i in the graph j, i.e., term frequency, and $\mathbf{idf}(i) = \log(\frac{N}{\mathbf{df}(i)})$, is the inverse document frequency, where N is the total number of graphs and $\mathbf{df}(i)$ is the number of graphs containing the term i. As discussed previously, the **idf** factor ensures that a common term (an API function node or an edge) which appears in a lot of graphs is not relevant (for example the API functions strlen, strcpy, strcat, etc. appear frequently in all the API call graphs but are not concerned with malicious behaviours).

F can be instanciated by the functions F_1, F_2, F_3 or F_4 defined in Subsection 7.1, where $\mathcal{D}$ is substituted by $\mathcal{G}$, $\mathbf{S}(j)$ is the size of graph j measured by the number of occurrences of terms in this graph [$\mathbf{S}(j)$ is the number of nodes (resp. edges) in the graph j if we are computing the weight of a node (resp. edge)]. $\mathbf{AVG}(\mathcal{G}) = \frac{\sum_{i=1}^{|\mathcal{G}|}\mathbf{S}(i)}{|\mathcal{G}|}$ is

the average size of graphs in $\mathcal{G}$. As previously, these F functions apply the optimisations that are made in the IR community in order to normalise graph sizes and to ensure that high **tf** for a relevant term in an extended API call graph does not place that graph ahead of other graphs which have multiple relevant terms but with lower **tf** values.

As mentioned previously, these F_i functions were considered in the IR community because they have shown good performances rather than any theoretical reason. Depending on the application, one function can be better than the others. Thus, we will apply all these 4 functions in our context and see which one gives the best results.

The relevance of a term i in $\mathcal{G}$ is measured by its relevance in each graph in this set: it is computed as the sum of the term weights of i in each graph of $\mathcal{G}$:

$$W(i,\mathcal{G}) = \frac{1}{K}\sum_{j=1}^{|\mathcal{G}|} w_g(i,j) \tag{53}$$

where $K = \max_{i,j} w_g(i,j)$ is a normalising coefficient. It is used to normalise the term weight values in the different graphs to make them comparable in the summation. $W(i,\mathcal{G})$ is the weight of term i in the set $\mathcal{G}$. A term with a higher weight is more relevant to graphs in set $\mathcal{G}$.

7.3 Term weights wrt. malicious and benign graphs

Let $\mathcal{G}_M$ and $\mathcal{G}_B$ be respectively the sets of malicious and benign extended API call graphs. Let i be a term (a node or an edge in these graphs). If i is relevant for both $\mathcal{G}_M$ and $\mathcal{G}_B$, then it is not relevant for us, as it does not correspond to a malicious behaviour. Thus, we need to extract terms that are relevant in $\mathcal{G}_M$ and not in $\mathcal{G}_B$. For this, we compute the relevance of i in both $\mathcal{G}_M$ and $\mathcal{G}_B$, and then we rank the relevance of i in $\mathcal{G}_M$ wrt. $\mathcal{G}_B$. Thus, for every term i, we need to compute a new weight that computes the relevance of a term i in set $\mathcal{G}_M$ with respect to its relevance in set $\mathcal{G}_B$. We apply two intuitive equations.

Firstly, we use the Rocchio equation (Manning and Raghavan, 2009), which computes the relevance of a term i in a set of graphs $\mathcal{G}_M$ against the other set $\mathcal{G}_B$ as follows:

$$W(i,\mathcal{G}_M,\mathcal{G}_B) = \beta \times \frac{W(i,\mathcal{G}_M)}{|\mathcal{G}_M|} - \gamma \times \frac{W(i,\mathcal{G}_B)}{|\mathcal{G}_B|} \tag{54}$$

where $|\mathcal{G}_M|$ and $|\mathcal{G}_B|$ are the sizes of the sets $\mathcal{G}_M$ and $\mathcal{G}_B$, β and γ are parameters to control the effect of the two sets $\mathcal{G}_M$ and $\mathcal{G}_B$. Parameters β and γ are typically set to 0.75 and 0.15, respectively (Manning and Raghavan, 2009). Thus, we take these values in our experiments. This means that the weight of term i in $\mathcal{G}_M$ has a high impact for its relevance weight.

The equation above computes the relevance of a term i as the distance between the weight of i in the set $\mathcal{G}_M$ and its weight in the set $\mathcal{G}_B$. A higher distance means that i is more relevant for $\mathcal{G}_M$ than for $\mathcal{G}_B$.

Secondly, we compute the relevance of a term i as follows:

$$W(i,\mathcal{G}_M,\mathcal{G}_B) = \frac{W(i,\mathcal{G}_M)}{|\mathcal{G}_M|} \times \frac{\lambda + |\mathcal{G}_B|}{\lambda + W(i,\mathcal{G}_B)} \tag{55}$$

Intuitively, this is computed as the weight of i in $\mathcal{G}_M$ divided by the weight of i in $\mathcal{G}_B$. Thus, if i has a high weight in $\mathcal{G}_M$ and a low weight in $\mathcal{G}_B$, it will have a high relevance weight. So, a higher weight indicates a bigger relevance of i in $\mathcal{G}_M$. The coefficient λ is added to avoid having a problem when $W(i, \mathcal{G}_B) = 0$. In the experiments, we set $\lambda = 0.5$. The above equation will be referred to as 'the ratio equation' in the rest of the paper.

As the two equations above (Rocchio and ratio) are natural and intuitive, and since there is no theoretical evidence that shows the advantage of taking one equation over the other, we compare the performances of these two equations in our experiments.

7.4 Computing the malicious extended API call graphs

A malicious extended API call graph is a tuple $G_M = (V_M, E_M, V_0, V_F)$, where V_M is the set of nodes, E_M is the set of edges, V_0 is the set of initial vertices, and V_F is the set of final nodes. It is a combination of terms which have a high malicious relevance. Since terms of a graph are either nodes or edges, we compute the malicious extended API call graph as follows. We consider a parameter n which is chosen by the user. n corresponds to the number of nodes that are taken into account in the computation. We show how to compute the sets V_M and E_M, the sets V_0 and V_F are the sets of nodes that do not have entering (resp. exiting) edges. In what follows, by 'weight of a term i', we refer to $W(i, \mathcal{G}_M, \mathcal{G}_B)$. We consider all the nodes and the edges in all the extended API call graphs of the collection. Let $\{v_1, \ldots, v_n\}$ be the set of vertices that have the n highest weights. Intuitively, this means that the vertices in $\{v_1, \ldots, v_n\}$ are the most relevant ones. Then, a natural way to compute the malicious extended API call graphs is to take the nodes in $\{v_1, \ldots, v_n\}$ and connect them using edges with the highest weight: Given a node v, if there are k outgoing edges $e_1, \ldots, e_k$ from v to nodes in $\{v_1, \ldots, v_n\}$, we add to the graph the edge e_j that has the highest weight.

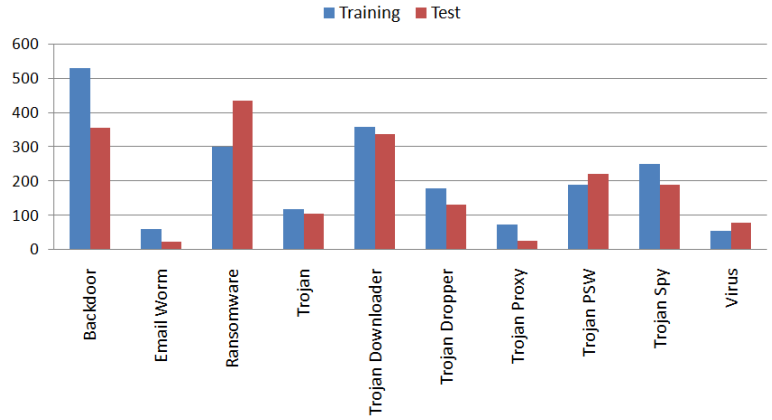
7.5 Malware detection using malicious graphs

Given a program represented by its extended API call graph $G = (V, E)$ and a set $\mathcal{M}$ of malicious behaviours represented by an extended malicious API call graph $G_M = (V_M, E_M, V_0, V_F)$, in order to check whether the program contains one of the malicious behaviours in $\mathcal{M}$, we need to check whether the two graphs G and G_M contain common paths. If this holds, we conclude that the program contains a malicious behaviour. To check whether the two graphs contain a common path, we compute a kind of product as follows: $G_P = (V_P, E_P, V_P^0, V_P^F)$ such that $V_P = \{(v_1, v_2) \in V \times V_M, func(v_1) = func(v_2) = f,$ and if $Para_M(f) \neq \emptyset$, then for every $a \in Para_M(f)$, $mean(v_1)(a) \cap mean\ (v_2)(a) \neq \emptyset\}$ is the set of vertices (v_1, v_2) such that v_1 and v_2 correspond to the same API function f, and for every meaningful parameter a of f, v_1 and v_2 require at least one common value for a. $V_P^0 = \{(v_1, v_2) \in V_P \mid v_2 \in V_0\}$, $V_P^F = \{(v_1, v_2) \in V_P \mid v_2 \in V_F\}$. $E_P = \{\big((v_1, v_2), (v_1', v_2')\big) \in V_P \times V_P \mid \exists (v_1, e_1, v_1') \in E, (v_2, e_2, v_2') \in E_M, e_2 \subseteq e_1\}$ is the set of edges $\big((v_1, v_2), (v_1', v_2')\big)$ such that there exist edges (v_1, e_1, v_1') in E and (v_2, e_2, v_2') in E_M such that the relation between the parameters of $func(v_1)$ and $func(v_1')$ imposed by the malicious behaviour in e_2 are satisfied by e_1. Then, the program contains a malicious behaviour in $\mathcal{M}$ iff G_P contains a path that leads from an initial node in V_P^0 to a final node in V_P^F.

8 Experiments

To evaluate the performance of our approach, we consider a dataset of 2,249 benign programs and 4,035 malicious programs collected from Vx Heaven (vxheaven.org) and from VirusShare.com. The proportion of malware categories is shown in Figure 2. We randomly divide the dataset in two partitions: one partition consisting of 1,009 benign programs and 2,124 malicious programs will serve as a training set that will allow us to compute the extended malicious API call graph, and the other partition consisting of 1,240 benign programs and 1,911 malicious programs will serve as a test set in order to evaluate the relevance of the computed malicious graphs. The experiements in this section are implemented on the laptop HP ZBook G2 Mobile Workstation – Intel Core i7-4710MQ 2.50 GHz and 8 GB RAM.

Figure 2 Malware categories in our dataset (see online version for colours)



First, we need to determine what function (amongst $F1, F2, F3$ and $F4$), and what equation (amongst the Rocchio and ratio equations [equations (54) and (55)] have the best performance). For each pair (function/equation), we construct the extended malicious API call graph from the training set. Then, these specifications are evaluated on this training set to choose the configuration which gives the best performance on our dataset. The performance is measured by the following quantities. True positives (TP) is the number of malware's extended API call graphs which contain at least one behaviour in the computed malicious extended API call graphs. False positive (FP) is the number of benign extended API call graphs which contain at least one behaviour in the computed malicious extended API call graphs. False positive rate (FPR or false alarm) is FP divided by the number of benign extended API call graphs. Recall (detection rate or TPR) is TP divided by the number of malware's extended API call graphs. Precision is TP divided by the sum of TPs and FPs. The F-measure is a harmonic mean of precision and recall that is computed as F-measure = $2 \times Precision \times Recall$ / $(Precision + Recall)$. These are standard evaluation measures in the IR community. High precision means that the technique computes more relevant items than irrelevant, while high recall means that most of the relevant retrieved items are relevant and all relevant items have been retrieved.

Figure 3 Results of different formulas on the training set, (a) Rocchio equation (b) ratio equation (see online version for colours)

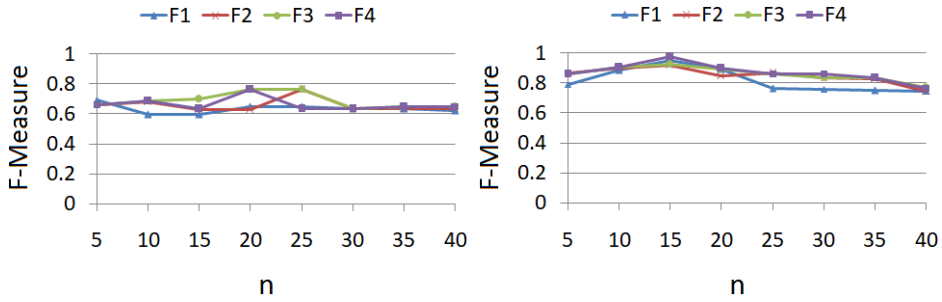


Figure 3 shows the F-measure for every weighting formula $F1, F2, F3$ and $F4$ by increasing n from 5 to 40. Figure 3(a) considers the Rocchio equation, whereas Figure 3(b) considers the ratio equation. We stop at $n = 40$ because we observe that the performance decreases for n greater than 40: the F-measure decreases, while the number of FPs increases. The results show that the best performance (F-measure = 97.78%, precision = 100%, and recall = 95.66%) is obtained if we consider formula $F4$, the ratio equation, and $n = 15$. Thus, we apply this configuration (ratio equation, formula $F4$ and $n = 15$) to compute the extended malicious API call graph from the training set. Then, we apply this graph for malware detection on the test set (see Subsection 7.5). We obtain a detection rate of 95.66% with 0 false alarms. Computing the extended malicious API call graph from the training set takes 12.61 seconds, whereas malware detection on the test set takes 0.394 seconds (average time).

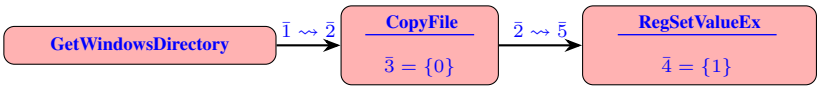
To compare the performance of our extended API call graph with that of standard API call graphs (in API call graphs, only API functions are considered, neither the values of parameters, nor the relation between function parameters are taken into account), we applied the techniques of Dam and Touili (2016) to our test set. We obtained 95.29% as detection rate, with 12.58% as false alarms, whereas with our approach, we get a detection rate of 95.66% with 0 false alarms. Thus, our extended API call graph representation and our approach allow an enormous decrease of the number of false alarms, and even improve a bit the detection rate. This shows the big advantage of using extended API call graphs over standard API call graphs.

9 Examples of malicious behaviours

Using our techniques, we were able to extract several extended API call graphs that represent meaningful malicious behaviours. For example, we were able to *automatically* extract the extended API call graph of Figure 1(c). We show in what follows other graphs that were *automatically* extracted using our techniques.

- *Windows directory infection:* The graph in Figure 4 describes the malicious behaviour that consists in replacing a system file by a malicious executable and then updating this change in the registry key listing.

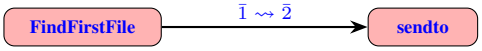
Figure 4 The extended malicious API graph of the malicious behaviour of Windows directory infection (see online version for colours)



First, the function `GetWindowsDirectory` is called to get the location of Windows directory in the system. This location is stored in the first parameter (the output of the function). Then, the function `CopyFile` with 0 as third parameter and the Windows directory as second parameter is called to replace the system file in the Windows directory by a malicious executable. Finally, the function `RegSetValueEx` is called with 1 as fourth parameter and the Windows directory (the second parameter of the previous function) as its fifth parameter to update this change of the system folder in the registry key listing.

- *Searching files and sending them via the network:* In Figure 5, the graph describes the malicious behaviour that consists in searching files on the system and sending them via the network.

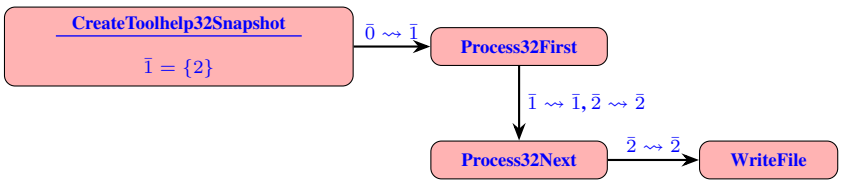
Figure 5 The extended malicious API graph of the malicious behaviour of searching files and sending them via the network (see online version for colours)



This malicious behaviour is implemented by first calling `FindFirstFile` with a specific filename as first parameter to look for this file on the system. Then, the function `sendto` with this filename as second parameter is called to transfer this file via the network.

- *Capturing system processes:* The graph in Figure 6 describes the malicious behaviour that consists in getting all system processes running on the system and then storing them in a file.

Figure 6 The extended malicious API graph of the malicious behaviour of capturing system processes (see online version for colours)



This malicious behaviour is implemented by first calling `CreateToolhelp32Snapshot` with 2 as first parameter to capture all system processes running on the system. Then, the function `Process32First` and `Process32Next` with the output ($\bar{0}$) of the previous function (all the processes

captured by the previous function) as first parameter are called to traverse all captured processes. The information about processes are saved by calling function `WriteFile` with information of processes (the second parameter of the previous function) as second parameter.

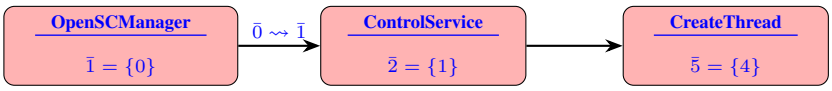
- *Dropping and executing a file from the internet:* The graph in Figure 7 describes the malicious behaviour that consists in downloading a file to Windows directory from the internet and then executing this file.

Figure 7 The extended malicious API graph of the malicious behaviour of dropping and executing a file from the internet (see online version for colours)

This malicious behaviour is implemented by first calling `GetWindowsDirectory` to get the Windows directory stored in the first parameter. Then, the function `URLDownloadToFile` with Windows directory (the first parameter of the previous function) as third parameter is called to download a file to this directory. Finally, the function `ShellExecute` with 0 as sixth parameter and the downloaded file as third parameter is called to open this file in the background.

- *Stopping the local service and starting a new thread:* In Figure 8, the graph describes the malicious behaviour that consists in stopping the security service in the system and then creating a new thread to handle malicious code.

Figure 8 The extended malicious API graph of the malicious behaviour of stopping the local service and starting a new thread (see online version for colours)

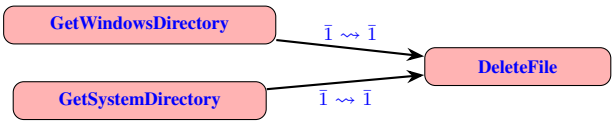


This malicious behaviour is implemented by first calling `OpenSCManager` with 0 as first parameter to get the service manager of the system. Then, `ControlService` with the output ($\bar{0}$) of the previous function (a handle of the secured service) as second parameter is called to stop the secured service in the system. Finally, `CreateThread` with 4 as fifth parameter is called to execute the malicious code.

- *Deleting files in the system folder:* The graph in Figure 9 describes the malicious behaviour that consists in deleting files in the Windows/system directory.

  The behaviour is implemented by first calling `GetWindowsDirectory` (resp. `GetSystemDirectory`) to get the Windows (resp. system) directory as first parameter. Then, calling `DeleteFile` takes the first parameter of the previous call [the path of Windows (resp. system) directory] as first parameter. With this parameter, this call deletes files in the Windows (resp. system) directory.

Figure 9 The extended malicious API graph of the malicious behaviour of deleting files in the system folder (see online version for colours)



10 Conclusions

In this paper, we propose to use extended API call graphs to obtain a more precise representation of malicious behaviours. In order to compute such graphs, we need to compute the relation between program's variables at the different locations of the program. To this aim, we propose a new translation from binary code to PDSs different from the standard one (Song and Touili, 2016), where we push a pair (x, n) into the PDS stack if at the location n, the variable x is pushed (whereas in the standard translation, only the value of x at location n is pushed in the PDS stack). Then, to compute the relation between the different variables and functions' arguments of the program, we present potentially infinite configurations of programs (PDSs) using finite automata, and we adapt the PDS $post^*$ saturation procedure of Esparza et al. (2000) in order to compute an annotation function from which we can compute the variable dependence function at each location in the program. This allows to compute the relation between program variables and function's arguments, and thus, to compute the extended API call graph of the program. Using this representation, we apply the IR techniques to compute the extended malicious API graph corresponding to malicious behaviours from a set of malwares and a set of benign programs. Compared to the API call graph representation, using the extended API call graphs improves the detection rate and the number of false alarms. Indeed, we obtain 95.66% as detection rate, with 0% false alarms for the extended API call graphs. Thus, the extended API call graph representation and our new approach in this paper allow an enormous decrease of the number of false alarms. This shows the big advantage of using extended API call graphs, compared to standard API call graphs.

References

Anderson, B., Quist, D., Neil, J., Storlie, C. and Lane, T. (2011) 'Graph-based malware detection using dynamic analysis', *Journal in Computer Virology*, Vol. 7, No. 4, pp.247–258, ISSN: 1772-9904, DOI: 10.1007/s11416-011-0152-x.

Baldangombo, U., Jambaljav, N. and Horng, S-J. (2013) *A Static Malware Detection System using Data Mining Methods*, arXiv preprint arXiv:1308.2831.

Bergeron, J., Debbabi, M., Erhioui, M.M. and Ktari, B. (1999) 'Static analysis of binary code to isolate malicious behaviors', in *Proceedings of the 8th Workshop on Enabling Technologies on Infrastructure for Collaborative Enterprises, WET ICE'99*, DOI: 10.1109/ENABL.1999.805197.

Bhatkar, S., Chaturvedi, A. and Sekar, R. (2006) 'Dataflow anomaly detection', in *2006 IEEE Symposium on Security and Privacy (SP'06)*, May, pp.15–62, DOI: 10.1109/SP.2006.12.

Bouajjani, A., Esparza, J. and Maler, O. (1997) 'Reachability analysis of pushdown automata: application to model-checking', in *International Conference on Concurrency Theory*, Springer, pp.135–150.

Canzanese, R., Mancoridis, S. and Kam, M. (2015) 'System call-based detection of malicious processes', in *2015 IEEE International Conference on Software Quality, Reliability and Security*, pp.119–124.

Cheng, J.Y-C., Tsai, T-S. and Yang, C-S. (2013) 'An information retrieval approach for malware classification based on windows API calls', in *2013 International Conference on Machine Learning and Cybernetics*.

Christodorescu, M. and Jha, S. (2003) 'Static analysis of executables to detect malicious patterns', in *Proceedings of the 12th Conference on USENIX Security Symposium, SSYM'03*, Vol. 12 [online] http://dl.acm.org/citation.cfm?id=1251353.1251365.

Christodorescu, M., Jha, S. and Kruegel, C. (2007) 'Mining specifications of malicious behavior', in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC-FSE'07*, ACM, ISBN: 978-1-59593-811-4, DOI: 10.1145/1287624.1287628.

Cyber Security Ventures (2017) *Ransomware Damage Report* [online] https://cybersecurityventures.com/ransomware-damage-report-2017-5-billion/ (accessed 25 August 2017).

Dam, K-H-T. and Touili, T. (2016) 'Automatic extraction of malicious behaviors', in *Proceedings of the 11th International Conference on Malicious and Unwanted Software (MALWARE)*, IEEE, pp.1–10.

Dam, K.H.T. and Touili, T. (2018) 'Precise extraction of malicious behaviors', in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, July, pp.229–234, DOI: 10.1109/COMPSAC.2018.00036.

Eagle, C. (2011) *The IDA Pro Book*, 2nd ed., No Starch Press, San Francisco, CA, USA.

Elhadi, E., Maarof, M.A. and Barry, B. (2013) 'Improving the detection of malware behaviour using simplified data dependent API call graph', *International Journal of Security and Its Applications*, Vol. 7, pp.29–42, DOI: 10.14257/ijsia.2013.7.5.03.

Esparza, J., Hansel, D., Rossmanith, P. and Schwoon, S. (2000) 'Efficient algorithms for model checking pushdown systems', in *International Conference on Computer Aided Verification*, Springer, pp.232–247.

Fredrikson, M., Jha, S., Christodorescu, M., Sailer, R. and Yan, X. (2010) 'Synthesizing near-optimal malware specifications from suspicious behaviors', *2010 IEEE Symposium on Security and Privacy*, pp.45–60, ISBN: 978-0-7695-4035-1, DOI: 10.1109/SP.2010.11.

Gavrilut, D., Cimpoesu, M., Anton, D. and Ciortuz, L. (2009) 'Malware detection using perceptrons and support vector machines', in *2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*, IEEE.

Kapoor, A. and Dhavale, S. (2016) 'Control flow graph based multiclass malware detection using bi-normal separation', *Defence Science Journal*, Vol. 66, p.138.

Khammas, B.M., Monemi, A., Bassi, J.S., Ismail, I., Nor, S.M. and Marsono, M.N. (2015) 'Feature selection and machine learning classification for malware detection', *Jurnal Teknologi*, Vol. 77.

Kinable, J. and Kostakis, O. (2011) 'Malware classification based on call graph clustering', *J. Comput. Virol.*, November, Vol. 7, No. 4, ISSN: 1772-9890, DOI: 10.1007/s11416-011-0151-y.

Kinder, J. and Veith, H. (2008) 'Jakstab: a static analysis platform for binaries', in Gupta, A. and Malik, S. (Eds.): *Computer Aided Verification*, Vol. 5123, ISBN: 978-3-540-70543-7.

Kinder, J., Katzenbeisser, S., Schallhart, C. and Veith, H. (2010) 'Proactive detection of computer worms using model checking', *IEEE Transactions on Dependable and Secure Computing*, October, Vol. 7, No. 4, ISSN: 1545-5971, DOI: 10.1109/TDSC.2008.74.

Kolter, J.Z. and Maloof, M.A. (2004) 'Learning to detect malicious executables in the wild', in *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ISBN: 1-58113-888-1, DOI: 10.1145/1014052.1014105.

Kong, D. and Yan, G. (2013) 'Discriminant malware distance learning on structural information for automated malware classification', in *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

Kruegel, C., Mutz, D., Valeur, F. and Vigna, G. (2003) 'On the detection of anomalous system call arguments', *Computer Security – ESORICS 2003*, pp.326–343, ISBN: 978-3-540-39650-5, DOI: 10.1007/978-3-540-39650-5_19.

Lin, C-T., Wang, N-J., Xiao, H. and Eckert, C. (2015) 'Feature selection and extraction for malware classification', *Journal of Information Science and Engineering*, Vol. 31, pp.965–992.

Macedo, H.D. and Touili, T. (2013) 'Mining malware specifications through static reachability analysis', in *European Symposium on Research in Computer Security*, Springer, pp.517–535.

Manning, H.S.C.D. and Raghavan, P. (2009) *An Introduction to Information Retrieval*, Cambridge University Press, New York, NY, USA.

Masud, M.M., Khan, L. and Thuraisingham, B. (2008) 'A scalable multi-level feature extraction technique to detect malicious executables', *Information Systems Frontiers*, Vol. 10, pp.33–45.

Nikolopoulos, S.D. and Polenakis, I. (2016) 'A graph-based model for malware detection and classification using system-call groups', *Journal of Computer Virology and Hacking Techniques*, pp.1–18, ISSN: 2263-8733, DOI: 10.1007/s11416-016-0267-1.

Ravi, C. and Manoharan, R. (2012) 'Malware detection using windows API sequence and machine learning', *International Journal of Computer Applications*, Vol. 43, pp.12–16.

Rieck, K., Holz, T., Willems, C., Düssel, P. and Laskov, P. (2008) 'Learning and classification of malware behavior', in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, pp.108–125.

Robertson, S. and Zaragoza, H. (2009) *The Probabilistic Relevance Framework: BM25 and beyond*, Now Publishers Inc.

Robertson, S.E., Walker, S., Jones, S., Hancock-Beaulieu, M.M., Gatford, M. et al. (1995) *OKAPI at TREC-3*, Vol. 109, NIST Special Publication SP.

Santos, I., Ugarte-Pedrero, X., Brezo, F., Bringas, P.G. and Gómez-Hidalgo, J.M. (2013) 'NOA: an information retrieval based malware detection system', *Computing and Informatics*, Vol. 32, No. 1, pp.145–174.

Schultz, M.G., Eskin, E., Zadok, E. and Stolfo, S.J. (2001) 'Data mining methods for detection of new malicious executables', in *Proceedings 2001 IEEE Symposium on Security and Privacy, S&P 2001*, DOI: 10.1109/SECPRI.2001.924286.

Schwoon, S. (2002) *Model-Checking Pushdown Systems*, PhD thesis, Technische Universität München.

Shafiq, M.Z., Tabish, S.M., Mirza, F. and Farooq, M. (2009) 'PE-miner: mining structural information to detect malicious executables in realtime', in Kirda, E., Jha, S. and Balzarotti, D. (Eds.): *Recent Advances in Intrusion Detection, RAID 2009, Lecture Notes in Computer Science*, Vol. 5758, Springer, Berlin, Heidelberg.

Singhal, A. and Kaszkiel, M. (2001) 'A case study in web search using trec algorithms', in *Proceedings of the 10th International Conference on World Wide Web*, ACM.

Singhal, A., Buckley, C. and Mitra, M. (1996) 'Pivoted document length normalization', in *Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, ACM.

Singhal, A., Choi, J., Hindle, D., Lewis, D.D. and Pereira, F. (1999) *AT&T at TREC-7*, NIST Special Publication SP.

Song, F. and Touili, T. (2013) 'LTL model-checking for malware detection', in *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, *TACAS*.

Song, F. and Touili, T. (2016) 'Model-checking software library API usage rules', *Software & Systems Modeling*, October, Vol. 15, No. 4, pp.961–985, ISSN: 1619-1374, DOI: 10.1007/s10270-015-0473-1.

Tahan, G., Rokach, L. and Shahar, Y. (2012) 'Mal-ID: automatic malware detection using common segment analysis and meta-features', *J. Mach. Learn. Res.*, April, Vol. 13, No. 1, ISSN: 1532-4435 [online] http://dl.acm.org/citation.cfm?id=2503308.2343677.

Xu, M., Wu, L., Qi, S., Xu, J., Zhang, H., Ren, Y. and Zheng, N. (2013) 'A similarity metric method of obfuscated malware using function-call graph', *Journal of Computer Virology and Hacking Techniques*, Vol. 9, No. 1, pp.35–47, ISSN: 2263-8733, DOI: 10.1007/s11416-012-0175-y.

Yao, J., Wang, J., Li, Z., Li, M. and Ma, W-Y. (2006) 'Ranking web news via homepage visual layout and cross-site voting', in *European Conference on Information Retrieval*.

Ye, Y., Li, T., Jiang, Q., Han, Z. and Wan, L. (2009) 'Intelligent file scoring system for malware detection from the gray list', in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.