Dynamic Behavior Matching: A Complexity Analysis and New Approximation Algorithms

Matthew Fredrikson[1], Mihai Christodorescu[2], and Somesh Jha[1]

University of Wisconsin – Madison[1], IBM Research – Hawthorne, NY[2]

Abstract. A number of advances in software security over the past decade have foundations in the *behavior matching* problem: given a specification of software behavior and a concrete execution trace, determine whether the behavior is exhibited by the execution trace. Despite the importance of this problem, precise descriptions of algorithms for its solution, and rigorous analyses of their complexity, are missing in the literature. In this paper, we formalize the notion of behavior matching used by the software security community, study the complexity of the problem, and give several algorithms for its solution, both exact and approximate. We find that the problem is in general not efficiently solvable, i.e. *behavior matching is NP-Complete*. We demonstrate empirically that our approximation algorithms can be used to efficiently find accurate solutions to real instances.

1 Introduction

The prevalence of malicious software, and the inability of traditional protection mechanisms to stop it, has led security researchers and practitioners to develop behavior-based techniques [2, 3, 6, 11, 12, 13, 15, 16, 17, 18, 19]. Unlike the syntax-based techniques used for years to detect the presence of known malicious code, behavior-based techniques observe the actions of potentially-malicious code, and attempt to match then against pre-defined specifications of malicious behavior. Conventional thinking suggests that this is a better way to detect threats, as it is much more difficult for malicious code developers to obfuscate the behavior of their software than it is to obfuscate its syntax in memory. Experimental results and practical experience have supported these claims [2, 12, 13].

This has made the technical problem of matching behavior specifications to run-time program behavior important. However, when one surveys the literature in this area [2, 6, 12, 16, 17, 18], one finds that formal descriptions of algorithms for this operation are largely missing. Furthermore, given the reported performance characteristics of existing matching techniques [2, 12, 16, 18], it seems to be widely assumed that behavior matching can be done efficiently, e.g. accounts of 3%-5% overhead on baseline program runtime are common. So, when we implemented a behavior matching algorithm for a project last winter [9], we did not expect to encounter any performance problems. To our surprise, we found that matching simple behavior graphs against pre-recorded traces of short (120 second) executions either exhausted available memory resources, or took several days to complete.

This led us to examine our algorithm in an effort to determine the cause of its apparent high complexity. We determined that in order to correctly match our behavioral specifications to realistic traces, backtracking on the potential mappings between specification components and trace entries was needed. Furthermore, we could not envision a scenario in which the algorithms discussed in the literature would *not* need to perform a similar kind of backtracking. This prompted us to perform the study reported in this paper: a detailed formal examination of the inherent complexity of the problem posed by matching behavior specifications to concrete execution traces, and a study of potential algorithms, exact and approximate, for doing so. This paper makes the following contributions:

– We present a general formulation of behavior matching that encompasses the most prevalent accounts in the literature (Section 2), and use it to show that behavior matching is an NP-Complete problem under the conservative assumption that one allows equality dependencies between events in the behavior specification (Section 2). Furthermore, our formulation is sufficiently generic to apply to arbitrary software behaviors, and thus relevant to specification needs in problems outside of security, such as runtime verification.
– We give two exact algorithms for performing behavior matching: one in direct terms of our formalism, and one from a reduction to SAT that allows practitioners to benefit from recent advances in SAT-solving technology (Section 3).
– We also present two approximation algorithms for behavior matching. One algorithm allows the user to bound the probability of false positives for a small trade-off in runtime complexity, and the other runs in time linear in the size of the trace.

The rest of this paper is organized as follows. Section 2 formulates the problem of behavior matching, and gives our main complexity result. Section 3 presents several algorithms for solving behavior matching instances. Section 5 discusses related work, and Section 6 provides concluding remarks.

2 Definitions and Problem Statement

First, we discuss the notion of software behavior that defines the basis of the matching problem. Any propositions we state have corresponding proofs in the technical report [8]. All of our formalisms make use of *terms* [1] $T(\Sigma, V)$, which denotes the set of all terms over the signature Σ of function symbols and V of variables. We also make use of the projection function $\pi_i(\cdot)$, which takes a tuple and returns its ith component.

Intuitively, dynamic behavior matching seeks to determine whether an observed *execution trace*, or sequence of observable facts emitted by a program, "fits" a pre-defined *behavior specification*, which can be thought of as a set of observable facts together with dependencies that describe necessary relations between the facts. In previous work, the dependencies generally encode either equality [11, 17], or some predicate over the data in the facts (e.g. `SubStr` [9] or `taint` [13]). In our work, the "fits" relation is made precise as a mapping between the facts in the specification and the facts in the trace, that properly accounts for the dependencies in the specification. This characterization naturally gives rise to a graphical structure, which is the basis for our notion of specification, called the *behavior graph* (Definition 1). This definition is meant to encompass as many of the relevant notions of behavior from the related work as possible, without introducing features that would make the matching problem more complex. In other words, it should be possible to translate our specifications into other formalisms found in the literature.

Definition 1 Behavior Graph. *A* behavior graph $\mathbf{G}$ *is a 5-tuple* $(A, E, a_0, \alpha, \beta)$*, where*

- A *is a set of states, and* $a_0 \in A$ *is an initial state.*
- $E \subseteq A \times A$ *is a set of directed edges between states such that* (A, E) *is a DAG.*
- $\alpha : A \to T(\Sigma, V)$ *is a total mapping from each state to a* Σ-terms *over* V*.*
- $\beta : E \to (V \to T(\Sigma, V))$ *is a total mapping from each edge to* $T(\Sigma, V)$-substitutions*.*

The signature Σ used in α and β corresponds to a set of observable events in the system. α maps states in $\mathbf{G}$ to observable events, with variables from V $(V \cap \Sigma = \varnothing)$ allowing variation in the substructure of events. β serves to constrain the *dependencies* between the states connected by $e \in E$. For example, if $\beta((a, b)) = [v_1 = v_2 + 1]$, then the behavior graph has the equality constraint $v_1 = v_2 + 1$ on the edge (a, b).

As an example, much of the existing literature is concerned with *system calls* and *taint tracking*. This work can be represented in our framework by treating each system call as a function symbol in Σ, and introducing a symbol $\texttt{taint}^{(2)}$ such that $\texttt{taint}(x, y)$ denotes that fact that x matches the taint label of y. Labels are constant symbols, and represent data provenance, which can correspond to a number of system entities such as network connections, memory regions, and files. An example of this is given in Example 1.

Example 1. Download-then-Execute. The behavior graph given by:

- $A = \{s_0, s_1\}$, $E = \{(s_0, s_1)\}$, $a_0 = s_0$, $\beta = \{(s_0, s_1) \mapsto [l_2 \mapsto l_1]\}$
- $\alpha = \{s_0 \mapsto \texttt{download}(\texttt{taint}(x, l_1)), s_1 \mapsto \texttt{execute}(\texttt{taint}(y, l_2))\}$

corresponds to the download-then-execute behavior. It is depicted in the following diagram:

$$\begin{tikzcd}
\boxed{s_0 : \texttt{download}(\texttt{taint}(x, l_1))} \arrow[r, "{[l_2 \mapsto l_1]}"] & \boxed{s_1 : \texttt{execute}(\texttt{taint}(y, l_2))}
\end{tikzcd}$$

In this figure, the label on the edge corresponds to the β-constraint, and the labels on states to the corresponding α-labels. Note the β-constraint $[l_2 \mapsto l_1]$

between the two states, which states that the taint label in the second state must be equal to that in the first, effectively requiring that the data that is executed have the same *taint label* as the data that was downloaded.

Now we define an *execution trace*, the other relevant data in the behavior matching problem. An execution trace is a sequence of ground terms, as there are no unknowns about events that have already occurred in the execution of a program.

Definition 2 Execution Trace. *An* execution trace $\mathbf{T} \in T(\Sigma, \varnothing)^*$ *is a finite-length sequence of ground Σ-terms that may contain repetitions , where we use* $\text{Range}(\mathbf{T})$ *to denote the set of terms in the sequence and* $\mathbf{T}(i)$ *to the* i^{th} *element. Each term in* $\mathbf{T}$ *corresponds to a concrete observation about the execution of some program, with the interpretation that for* $i < j$, $\mathbf{T}(i)$ *occurs before* $\mathbf{T}(j)$ *in the program execution.*

The most common type of trace for behavior matching in the security literature is that obtained by letting Σ denote system calls and their arguments, often with meta-symbols for additional functionalities such as provenance and taint tracking, timing information, etc.

We now come to the primary definition of this section – behavior matching. Behavior matching is a problem defined in terms of a behavior graph and an execution trace; the goal is to determine whether the execution trace exhibits the behavior specified in the behavior graph. To simplify notation, in what follows we write $\beta_{x,y}$ to denote $\beta(x, y)$ and ρ^x_y for $\pi_x(\rho(y))$.

Definition 3 Behavior Matching. *Given a behavior graph* $\mathbf{G} = (A, E, a_0, \alpha, \beta)$ *and execution trace* $\mathbf{T}$*, we say that* $\mathbf{G}$ matches $\mathbf{T}$*, written* $\mathbf{G} \models \mathbf{T}$*, iff there exists a total function* $\rho : A \to \mathbb{Z}^+ \times (V \to T(\Sigma, V))$ *such that:*

1. $\rho_{a_0} = (i, \sigma)$*, where* $\sigma(\alpha(a_0)) = \mathbf{T}(i)$.
2. *For each* $(a, a') \in E$*:*

$$\rho^2_{a'}(\rho^2_a(\beta_{a,a'}(\alpha(a')))) = \mathbf{T}(\rho^1_{a'})$$

   Intuitively,
   - $\beta_{a,a'}(\alpha(a'))$ *is the term associated with the latter state* a'*, with dependencies instantiated according to* $\beta_{a,a'}$.
   - *Each application of* ρ^2_a *and* $\rho^2_{a'}$ *specializes the dependencies according to the trace terms to which* ρ *associates* a *and* a'*, respectively.*
3. *If* $i < j$ *and* $\rho_a = (i, \sigma), \rho_{a'} = (j, \sigma')$*, then* a *must be an ancestor of* a' *in* (A, E)*. Intuitively* ρ *maps states in* $\mathbf{G}$ *to terms in* $\mathbf{T}$ *that obey the temporal constraints introduced by the edges in* $\mathbf{G}$.
4. *For any* $a, a' \in A$, $\rho^1_a \neq \rho^1_{a'}$*, i.e.* ρ *cannot map two states to the same trace element.*

We call ρ a *witness* of the matching between $\mathbf{G}$ and $\mathbf{T}$. Intuitively, ρ maps a path through $\mathbf{G}$ to a sequence of ground terms in $\mathbf{T}$, while satisfying all temporal and data dependencies stipulated by the edges in $\mathbf{G}$.

Note that at times we abuse notation slightly by taking substitutions over terms, even though they are technically defined over variables; this is taken to mean the extension of the substitution over all free variables in the term. Also notice that in Definition 3, ρ must be a total mapping over A: all states in $\mathbf{G}$ must map to a term in $\mathbf{T}$ for the witness to be valid. Some researchers work with behavior graphs that have so-called "or-edge sets", which allow a matching trace to cover a path over one edge in the set, instead of all of them. This style of disjunctive behavior graph can be simulated with multiple graphs from Definition 1, one behavior matching instance per graph.

Example 2. The sequence

$$\texttt{download}(\texttt{taint}(\texttt{/tmp/data}, \texttt{l}_1)), \texttt{open}(\texttt{taint}(\texttt{/tmp/data}, \texttt{l}_1)),$$
$$\texttt{execute}(\texttt{taint}(\texttt{/tmp/data}, \texttt{l}_1))$$

matches the behavior graph from Example 1, with a witness that unifies the first and third terms in the trace with the behavior graph:

$$\left\{ s_0 \mapsto (1, [x \mapsto \texttt{/tmp/data}, l_1 \mapsto \texttt{l}_1]), s_1 \mapsto (3, [y \mapsto \texttt{/tmp/data}, l_2 \mapsto \texttt{l}_1]) \right\}$$

Example 3. The sequence

$$\texttt{download}(\texttt{taint}(\texttt{/tmp/data}, \texttt{l}_1)), \texttt{open}(\texttt{taint}(\texttt{/tmp/data}, \texttt{l}_1)),$$
$$\texttt{execute}(\texttt{taint}(\texttt{/bin/bash}, \texttt{l}_2))$$

does not match the behavior graph from Example 1. Looking to Definition 3, we see that the only way to unify $\alpha(s_0)$ with a term in the trace is

$$\sigma = [x \mapsto \texttt{/tmp/data}, l_1 \mapsto \texttt{l}_1]$$

So $\rho(s_0) = (1, [x \mapsto \texttt{/tmp/data}, l_1 \mapsto \texttt{l}_1])$. This gives us

$$\rho^1_{s_0}(\beta_{s_0,s_1}(\alpha(s_1))) = \texttt{execute}(\texttt{taint}(y, \texttt{l}_1))$$

There is no substitution that can unify this term with $\texttt{execute}(\texttt{taint}(\texttt{/bin/bash}, \texttt{l}_2))$, because of the mismatched taint labels.

We now move on to define two notions of behavior matching that specify different aspects of accuracy. Intuitively, soundness relates to false negatives, or the ability of an algorithm to correctly identify a matching execution trace when it is present, and completeness relates to false positives, or the ability of an algorithm to correctly identify traces that do not match a given graph.

Definition 4 Sound and Complete Matching Algorithm. *A matching algorithm* $\mathbf{A}$ *is a decision procedure for Definition 3. An algorithm* $\mathbf{A}$ *is a* sound matching algorithm *iff given a behavior graph* $\mathbf{G}$ *and trace* $\mathbf{T}$*,* $\mathbf{G} \models \mathbf{T} \Rightarrow \mathbf{A}(\mathbf{T}, \mathbf{G}) = \mathsf{True}$*. It is a* complete matching algorithm *iff* $\mathbf{A}(\mathbf{G}, \mathbf{T}) = \mathsf{True} \Rightarrow \mathbf{G} \models \mathbf{T}$.

Next, we discuss one of the central results of this work, which is that the inherent complexity of the behavior matching problem makes it intractable for most settings. To our knowledge, this is the first result of its kind for the problem.

Proposition 1 *Sound and complete behavior matching, with plain equality constraints between states, is NP-complete.*

By *plain* equality constraints, we are referring to dependencies that map a variable to another variable, without involving additional term structure. The proof [8] shows that checking a witness against a trace is a polynomial operation, and reduces instances of sub-DAG isomorphism (previously shown to be NP-Complete [20]) to behavior matching. The reduction treats nodes and edges in each of the DAGs as though they are events in Σ. For the larger of the DAGs, all of the structure is encoded in constant symbols, and the reduction views it as an execution trace. The reduction encodes the structure of the smaller of the two DAGs as dependence relations, and produces a corresponding behavior graph. Notice that the reduction only uses simple equality dependencies in the behavior graph; this implies that even the most simple dependencies arising in behavior graphs can lead to intractable instances of the problem. An example reduction on small graphs is given in Example 4.

Example 4. Consider the subgraph isomorphism problem given in Figure 1(a), where we would like to determine whether the three-node graph is isomorphic to a subgraph of the four-node graph. We reduce this to the instance of behavior matching given in Figure 1(b). Beginning and ending sentinels, `s` and `f` respectively, are added to the trace. In the reduction of the smaller graph to a behavior graph, nodes are represented by states that have corresponding $\mathtt{n}(x)$ terms under α, and edges to states with $\mathtt{e}(x_1, x_2)$ terms. Data dependencies are introduced to reflect the fact that the arguments of the terms on edge states must match the arguments of the corresponding node-state endpoints, as shown on the edges of the behavior graph in Figure 1 (b). The matching problem has a witness:

$$\left\{\begin{array}{l} \rho(s_0) = (1, \varnothing), \rho(s_6) = (10, \varnothing), \\ \rho(s_1) = (2, [x_1 \mapsto \mathtt{o}_1]), \rho(s_4) = (5, [x_6 \mapsto \mathtt{o}_2]), \\ \rho(s_5) = (7, [x_7 \mapsto \mathtt{o}_3]), \rho(s_2) = (2, [x_2 \mapsto \mathtt{o}_1, x_3 \mapsto \mathtt{o}_2]), \\ \rho(s_3) = (4, [x_4 \mapsto \mathtt{o}_1, x_5 \mapsto \mathtt{o}_3]) \end{array}\right\}$$

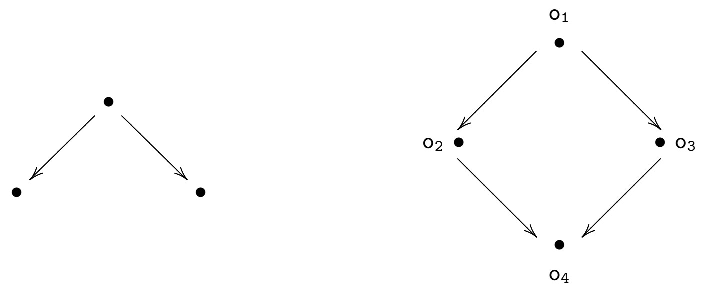
This mapping gives an isomorphism for the original subgraph isomorphism problem: the top three nodes in each graph map to each other.

3 Algorithms

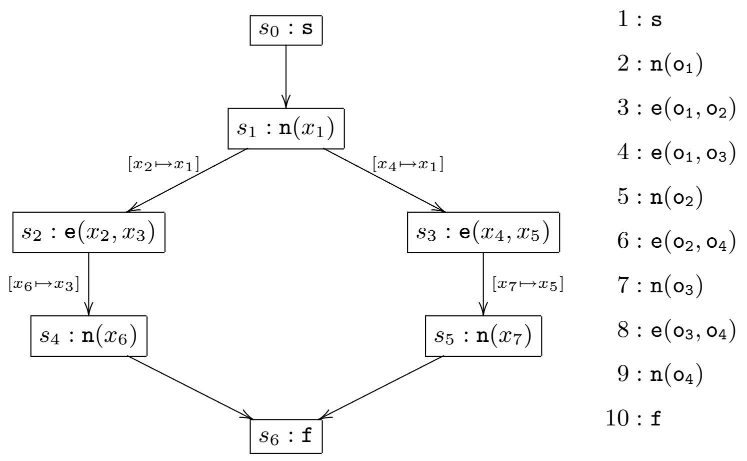
In this section, we detail several solutions for solving instances of the behavior matching problem. We begin with an exact algorithm, and conclude our discussion with two sound approximation algorithms.

Preliminaries. All of our algorithms are built from a shared collection of entities and primitives, which vary in detail among different algorithms:

- $\mathcal{F}$, the *frontier set*, which represents the current state of the matching operation. Different algorithms will place different sorts of elements in $\mathcal{F}$, but we



(a) Sub-DAG isomorphism instance.



(b) Corresponding reduction to behavior matching.

Fig. 1: Sub-DAG isomorphism to behavior matching reduction.

always use Γ to refer to the sort of elements in $\mathcal{F}$. The frontier set is analogous to a similar notion used in classic algorithms for matching NFAs [10], which consists of the set of states that the matching as reached at a given time.

- Test : $A \times \mathbb{Z}^+ \times \Gamma \mapsto \{\mathsf{True}, \mathsf{False}\}$, returns true if and only if the given frontier element (3rd argument) can be extended by matching the given state (1st argument) to the trace element at the given index (2nd argument).
- Update : $\mathcal{P}(\Gamma) \times A \times \mathbb{Z}^+ \times \Gamma \mapsto \mathcal{P}(\Gamma)$, updates the given frontier set (1st argument) to reflect an extension to the given frontier element (4th argument) by a mapping between the state/indexed trace term pair (2nd and 3rd arguments), returning a new frontier.

These primitives are used in Algorithm 1 (explained below), and are later modified to obtain approximation algorithms.

A Sound and Complete Algorithm. Algorithm 1 presents a sound and complete behavior matching algorithm. This property is a result of the definitions

Algorithm 1 Exact behavior matching

```
1: Input: Behavior graph $\mathbf{G} = (A, E, a_0, \alpha, \beta)$, execution trace $\mathbf{T}$
2: // Each element in the frontier maps states to trace terms and substitutions
3: $\Gamma = A \mapsto (\mathbb{Z}^+, V \mapsto T(\Sigma, V))$
4: // Test$(\cdot,\cdot,\cdot)$ Applies the dependence requirement for each edge $(s', s)$ on $s$
5: Test$(s, i, w) \equiv \exists\sigma.(\forall(s', s) \in E.\sigma(w^2_{s'})(\beta_{s',s}(\alpha(s)))) = \mathbf{T}(i))$
6: // Update$(\cdot,\cdot,\cdot,\cdot)$ extends witness $w$ with a mapping from $s$ to $\mathbf{T}(i)$
7: Update$(\mathcal{F}, s, i, w) = \mathcal{F} \cup \{w^1[s \mapsto (i, \sigma)]\}$ where $\sigma$ matches $\alpha(s)$ to $\mathbf{T}(i)$
8: $\mathcal{F} \leftarrow \varnothing$
9: for $0 \le i \le |\mathbf{T}|$ do
10:   for $w \in \mathcal{F}$ do
11:     for each next matchable state $s$ on frontier element $w$ do
12:       // Test dependencies with predecessors in current derivation
13:       if Test$(s, i, w)$ = True then
14:         // Extend frontier with matching term and updated derivation
15:         $\mathcal{F} \leftarrow$ Update$(\mathcal{F}, s, i, w)$
16:         if $w$ is a complete mapping then
17:           return (True, $w$)
18:         end if
19:       end if
20:     end for
21:   end for
22:   // If current trace term matches the initial state of $\mathbf{G}$, update frontier
23:   if Test$(s, i, \varnothing)$ = True then
24:     $\mathcal{F} \leftarrow$ Update$(\mathcal{F}, a_0, i, \varnothing)$
25:   end if
26: end for
27: return False
```

on lines 2 − 7, which specify the core primitives of the algorithm. The first definition is of Γ, the sort of element found in $\mathcal{F}$. Γ corresponds to mappings from states in $\mathbf{G}$ to pairs of trace indices and substitutions (e.g. *partial witnesses*). The substitution in this pair unifies the term associated with the state with the trace element indexed by the first component. For example, if $\gamma \in \Gamma$, then $\mathbf{T}(\gamma^1_a)$ matches $\gamma^2_a(\alpha(a))$. By defining Γ in this way, the algorithm can build a full set of possible partial witnesses as it enumerates a trace, and return a complete witness if a match exists; no approximation is necessary. Notice the correspondence between this definition of Γ (and thus $\mathcal{F}$) to the frontier used in traditional NFA matching: partial witnesses represent possible intermediate states as $\mathbf{G}$ is matched with $\mathbf{T}$.

The second definition is of Test, on line 5. This definition is a formal restatement of the dependence relation stated in Definition 3. When Test(s, i, w) is applied, all predecessors of s in $\mathbf{G}$ are checked, according to the mappings in the given witness w, for satisfaction of the dependence requirement stated in Definition 3. The expression is True iff the requirement holds for all predecessors of s. The final definition, Update$(\mathcal{F}, s, i, w)$, extends the witness w with a mapping from state s to trace element $\mathbf{T}(i)$, and the substitution σ that makes it possible. Notice that this definition of $\mathcal{F}$ does not drop or replace elements, but only adds new elements that are extensions of existing ones. This is equivalent to allowing the algorithm to backtrack, and is the source of the algorithm's complexity. If partial witnesses were dropped once their frontiers were all matched, it would make the algorithm "greedy", as backtracking would become impossible. However, it would also make the algorithm unsound, as it would open up the possibility for a trace to "trick" the algorithm into going down a particular path, which never leads to a full witness, and then pursuing an alternate matching path from midway through the original path. The corresponding technical report [8] describes this issue in more depth.

The rest of Algorithm 1 works by scanning the behavior trace $\mathbf{T}$ from beginning to end, one element at a time. For each trace element, each frontier element is enumerated (line 10), and an attempt is made to match the trace term to a term corresponding to the next matchable states of the frontier element (line 13). When the algorithm starts, the frontier is empty, so the matching on line 23 attempts to pair trace terms with the initial state in $\mathbf{G}$. If the initial-state matching succeeds, then the frontier is updated (line 24) to reflect this partial matching. In subsequent iterations, an attempted match between the latest trace term $\mathbf{T}(i)$ and each element in the frontier (line 15) is made, and if successful, the frontier is extended (line 15). This is continued until a complete witness is constructed, at which point the algorithm returns True (line 17).

Proposition 2 *The worst-case complexity of Definition 1 is $O(|A|MD^{|\mathbf{T}|})$ in time and $O(D^{|\mathbf{T}|})$ in space, where D is the maximal out-degree for any state in a given behavior graph $\mathbf{G}$ and execution trace $\mathbf{T}$, and M is the maximum time needed to match terms in $T(\Sigma, V)$.*

The operation of reference in Proposition 2 is the term matching between terms on states in $\mathbf{G}$ and events in $\mathbf{T}$. Note that because our term alphabet is finite, there is a hard upper bound on term size, and thus on the complexity of term matching. The technical report has a proof of soundess and completeness for Definition 1, as well as a proof of Proposition 2.

Sound Approximation Algorithms. The worst-case time and space complexity of Algorithm 1 make it a poor fit for many applications, particularly those involving long-running applications. In this section, we discuss approximation algorithms that mitigate this issue. We are only interested in sound approximation algorithms that never fail to detect a matching trace, but may spuriously decide that a benign trace matches a behavior graph. This property maintains the crucial guarantee that the algorithm will detect all attacks defined by the behavior graphs. We view this as the most important property that a behavior matching algorithm can possess, provided that the false positive rate is not unreasonably high.

The first approximation algorithm is obtained by re-defining the primitives $\mathcal{F}$, Test, and Update in Algorithm 1; the formal definitions of the primitives for this approximation algorithm are given in the technical report [8]. It performs matching by maintaining a memory, for each state in $\mathbf{G}$, of terms from $\mathbf{T}$ that may be matched to that particular state. If at any point a substitution exists that matches a frontier element to a trace element, the algorithm considers each predecessor s' of s, and checks the memory for each one to determine whether previous trace terms satisfy the dependencies needed to match $\mathbf{T}(i)$ to $\alpha(s)$. If so, then the frontier is updated with the new matching between s and $\mathbf{T}(i)$, and the next trace event is considered.

The imprecision in this algorithm comes from the fact that the frontier does not record partial witnesses, but instead only *local* history with respect to each states in $\mathbf{G}$. This means that Test might consult substitutions that would belong to multiple distinct partial witnesses in the exact algorithm, thus incorrectly concluding that all dependence constraints are satisfied when a single witness that satisfies all constraints does not exist. While this can lead to false positives, note that if a true witness does exist, then Test will effectively find it, so there can be no false negatives. This is related to existential abstraction, which describes the relationship between these approximate primitives, and the exact ones listed in Algorithm 1.

The next proposition guarantees that the amount of work required by the algorithm on receiving a new trace element is at most linear in the size of the behavior graph, length of the execution trace,and maximum term size of Σ.

Proposition 3 *When the approximate primitives are used in Algorithm 1, each iteration of the main loop is $O(iM|A|)$, where i is the current index into $\mathbf{T}$ and M is the maximum time needed to match terms in $T(\Sigma, V)$.*

Note that Proposition 3 implies that the algorithm has a worst-case time complexity that is linear in $|\mathbf{T}||A|$. The worst-case space complexity is $O(|\mathbf{T}|)$, as the frontier requires exactly one entry for each trace term.

We now present an approximation algorithm that uses Bloom filters [4] to record possible matchings between states in $\mathbf{G}$ and terms in $\mathbf{T}$. Intuitively, a set of Bloom filters is kept for each argument of each term in the image of α. As trace terms are matched to states, the corresponding substitutions for arguments are added to the filters, and later consulted when dependencies are matched. This algorithm has linear complexity with a very low coefficient (see Proposition 4), but at the cost of increased false positives due to the overapproximation of the Bloom filters. We represent the domain of Bloom filters by the symbol $\mathcal{B}$. The algorithm is obtained by substituting the following primitives in Algorithm 1:

- $\Gamma = (A \mapsto (\mathbb{Z}^+ \mapsto \mathcal{P}(\mathcal{B})), A)$. The first component of an element in Γ is a mapping from states to a different set of mappings, that contain an overapproximation for each term argument previously bound to the state. Note that the structure used for this overapproximation (the range of the mapping $\mathbb{Z}^+ \mapsto \mathcal{P}(\mathcal{B})$) is a set of Bloom filters, rather than a single Bloom filter. We override the default Bloom filter union operation to add a new filter to this set when the probability of encountering a false positive in the filter exceeds

ϕ. Similarly, the membership query operation must be overridden to check *all filters in the set* to maintain soundness.

– $\mathsf{Test}(s, i, w)$ returns True whenever:

$$\forall (s', s) \in E.\ \forall 0 \le j \le \text{arity}(\alpha(s')).\exists f \in w^1_{s'}(j).$$
$$[\text{args}(\alpha(s'), j) \mapsto f](\beta_{s',s}(\alpha(s))) \text{ matches } \mathbf{T}(i)$$

In other words, the Bloom filters associated with all predecessors of s are checked for elements that satisfy the needed dependencies.

– Update,

$$\mathsf{Update}(\mathcal{F}, s, i, w) = \mathcal{F} - \{w\} \cup$$
$$(w^1[s \mapsto \delta_s], w^2 \cup \{a : (a, s) \in E\})$$

where

$$\delta_s(k) = w^1_k \cup \text{args}(\alpha(s), k)$$

In other words, all of the Bloom filters associated with s are updated to reflect the arguments of $\mathbf{T}(i)$, and the old w is removed from $\mathcal{F}$.

We note that it is not strictly necessary to use a *set* of Bloom filters for each argument, particularly when the length of the inputs are known in advance, and the parameters of the Bloom filter can be configured to avoid false positives. This has the benefit of producing a constant-time algorithm in the size of the input trace. In many cases, however, it is not possible to determine a bound on trace length in advance. By using an unbounded set of Bloom filters to represent each argument, the algorithm overcomes the risk of encountering an explosion of false positives when the trace length exceeds the parameters of each individual Bloom filter. This is accomplished by overriding the union and membership query operations over filters in the definition of Test and Update, essentially tracking the number of entries inserted into each filter, and creating a new one when the probability of encountering a false positive rises above a user-specified value ϕ. When the history is consulted to establish a dependence, each of these filters must be checked to maintain soundness, and the number of the filters is a linear function of $|\mathbf{T}|$, thus the linear complexity in trace length.

Proposition 4 *The number of Bloom filters needed to maintain a false positive rate of at most ϕ is $O\left(-2|\mathbf{T}|k/2(b-1)\ln(1-\phi^{1/k}) + k\right)$, where k is the number of hash functions used in the Bloom filters, and b is the number of bits.*

Proposition 4 tells us that the complexity of the algorithm grows very slowly in the length of $\mathbf{T}$. Recall that the algorithm's complexity depends on $|\mathbf{T}|$ only insofar as the number of Bloom filters needed to maintain a false positive rate of no more than ϕ must be checked each time a new trace element is encountered. Proposition 4 tells us that this dependence is linear, with the given coefficient. Because the logarithm function has an asymptote at zero towards negative infinity, $-k/(b-1)\ln(1-\phi^{\frac{1}{k}})$ shrinks rather well in b and k, leaving the linear coefficient quite small. For example, devoting one megabyte of memory to the

Bloom filter ($b = 8,388,608$) and using $k = 100$ hash functions, the coefficient is approximately 2×10^{-6} for a false positive rate of no more than 1%. Needless to say, this is an impressive performance characteristic for a small amount of memory and imprecision.

4 Experimental Results

We performed experiments to determine the run-time characteristics of each of the algorithms presented in Section 3, in addition to the false positive rates of the approximation algorithms. We also implemented a reduction of behavior matching to SAT constraints (details in the technical report [8]), in order to evaluate the feasibility of using an off-the-shelf solver [7] for real instances of the problem. Our results are encouraging:

- The approximation algorithms perform significantly better than the exact algorithm; on our set, they performed 17.3 and 21.6 times faster (for the first and second approximation algorithms discussed, respectively), on average.
- The false positive rate of the approximation algorithms is not excessive: 7.3% and 9.1%.
- The SAT constraints corresponding to our data set are quickly solved by modern solvers, requiring 0.18 seconds to solve an instance, on average. However, generating the constraints generally requires a substantial amount of time and space: we observed on average 65.6 seconds and approximately 10^6 constraints for 120 second execution traces.

These results demonstrate the practical value of our algorithms.

We collected behavior traces from 70 applications (both known malware and common desktop applications), and matched them against ten behavior graphs mined from a repository of behavior data using simulated annealing [9]. The behavior traces are composed of system call events, along with detailed data and annotations about the arguments of each event; for an in-depth account of our behavior collection mechanism, consult our previous work [9, 13]. We ran all experiments on a quad-core workstation, with 8 gigabytes of main memory. For experiments involving Bloom filters, we utilized the `pybloom` library, with a low false positive probability ($\phi = 1\%$) and a moderate number of bits ($b = 4,000$).

The first noteworthy result we obtained is that the exact behavior matching algorithm presented in Definition 1 is significantly slower than both approximation algorithms, as well as the reduction to SAT constraints. On average, the precise algorithm required 31.35 seconds to complete, compared to 1.84 and 1.47 for the first and second algorithms discussed, respectively. The runtime overhead, which in this case corresponds to the amount of time taken by the algorithm taken as a percentage of the trace execution time,of the exact algorithm amounts to 26%, compared 1.5% and 1.2% for the approximation algorithms. Furthermore, 2% of the instances given to the precise algorithm timed out after 45 minutes. This confirms our suggestion that existing algorithms for behavior matching, which are purported to resemble Algorithm 1, have much higher complexity than previously thought.

Of the reduction to SAT, we found that while the instances can generally be solved quickly (more quickly than all of our algorithms, in fact), the constraint systems for an instance also grow quickly, and take a non-trivial amount of time to generate. In other words, the reduction to SAT is not yet suitable for runtime behavior matching, but may be ideally suited to off-line forensic analysis where exact solutions are required. We solved instances of SAT constraints using Minisat [7], which required on average 0.18 seconds to complete. We take this result as indication that it is common to encounter behavior matching instances that are "easy" in some sense. The running time of constraint generation on our dataset is distributed bimodally, with means at 16.6 seconds (87% of samples) and 883.7 seconds (6% of samples), and 7% timing out after 1 hour. This means that for the "easy" cases, the runtime overhead of constraint generation is approximately 14%, but for the "hard" cases, it is approximately 733%. The number of clauses in the constraint system has a similar bimodal distribution, with means at 6.7×10^5 (89% of samples) and 1.1×10^7 (4% of samples) clauses. If we assume that each clause takes 5 bytes of memory (a conservative underapproximation), then this means that on average, running an application for 120 seconds generates 3 megabytes of constraint data for easy cases, and 52 megabytes for hard cases; clearly, even for easy cases this does not scale to long-running applications.

Finally, we studied the false positive rates of the approximation algorithms, using the results of our precise algorithms (both Algorithm 1 and the SAT reduction) as ground truth. We found the rates to be reasonable: 7.36%, 9.13% for the first and second algorithms discussed, respectively. For the exceptionally low overhead produced by these methods (1.5% and 1.2%), we assert that this is an acceptable trade-off.

5 Related Work

Several abstractions with similarities to behavior graphs have been previously studied. Neven *et al.* studied both register and pebble automata [14] (RA and PA, respectively), which are two generalizations of traditional FSA to infinite alphabets. They conclude that PA is a more natural extension of FSA to infinite alphabets, but we do not see a way to encode a behavior graph as either formalism. The main issue is the bounded number of pebbles (or registers), which must be used to calculate dependencies; because a graph state may need to be temporarily matched to an unbounded number of trace events, the bounded number of pebbles (or registers) will cause the matching algorithm to drop history.

Another related formalism that has recieved attention is tree automata [5], which operate over ranked alphabets. Behavior graphs cannot be reduced to traditional finite-state tree automata for two reasons: dependencies between subterms cannot be represented, and the set of initial states in the automaton must be finite, whereas the execution traces that serve as inputs are unbounded, and would therefore require an infinite set of possible initial states. Extensions of tree automata involving dependence relations between subterms have been studied

([5] Chapter 4), but not their extension to infinite-state automata, which would be required for direct application to the problem of behavior matching.

Behavior matching has seen mention in the system security literature frequently in recent years. Perhaps the most compelling account is due to Kolbitsch *et al.* [12]. In this work, the authors describe behavior specifications that are nearly identical to those formalized in this paper, with nearly arbitrary data dependencies between events. An algorithm for matching execution traces to these specifications is alluded to, but a precise description of this algorithm is not given, much less an analysis of its complexity. The same notion of behavior and matching seems to be operative in other work by the same authors [2, 6]. However, the technique is pitched as *efficient* throughout the work, and reported overheads typically range around 5%. Given the strong connection between our notion of behavior matching and that presented by Kolbitsch *et al.*, these results diverge significantly from the theoretical results presented in this paper, as well as the observed performance characteristics of the sound and complete algorithm.

Sekar and Uppuluri [17] discuss the use of *extended finite-state automata* (EFSA) in intrusion detection. EFSA bear resemblance to the behavior graphs discussed in this paper insofar as they allow general data dependencies (including equality), but the authors do not attempt to formalize the computational model that these dependences may adopt. An algorithm for run-time matching of EFSA is given, and the authors claim that the amount of work on receiving an event is $O(N)$, where N is the number of states in the EFSA. This conflicts with our results. However, this claim is given without proof, and seems to be predicated on the assumption that the algorithm needs only remember a bounded number of possible matching configurations (the authors state this assumption in the description of their algorithm). This indicates that their algorithm is a greedy version of Definition 1, and therefore unsound; this conclusion is backed by the complexity results presented in Section 3.

Tokhtabayev *et al.* describe a behavior matching scheme based on colored Petri nets [18]. The formalism used to describe behaviors shares nearly all of its salient features with our notion of behavior, including complex data dependencies. The performance overheads they report fall below 5%, but the complexity of their matching algorithm is not discussed, and a formal description of the algorithm is not given due to "limitations." There are several other accounts of behavior matching involving data dependencies in the security literature [9, 11, 13, 15, 16, 21] that use notions of behavior for various ends. There is also work that formalizes software behaviors in terms of events without accounting for data dependencies [3]; this work is interesting in contrast to ours, the simpler notion of behavior may be more suitable for certain applications.

6 Conclusion

In this paper, we presented a formulation of behavior matching that encompasses most of those seen in the literature, and demonstrated the problem is

NP-Complete. We proceeded to give two exact algorithms for solving the problem, presented two approximation algorithms, and demonstrated that they can be used to find accurate solutions to real instances of behavior matching. In the future, it will be important to determine whether real applications of behavior matching can be made to fit into a tractable subclass of the general problem presented here.

Acknowledgments We would like to thank Eric Bach, Drew Davidson, Bill Harris, and Thomas Reps for their helpful comments. The first author of this work is supported by a Microsoft Research PhD Fellowship. A portion of this work was supported by the DARPA CRASH program.

Bibliography

[1] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[2] U. Bayer. *Large-Scale Dynamic Malware Analysis.* PhD thesis, Technical University of Vienna, 2009.

[3] P. Beaucamps, I. Gnaedig, and J.-Y. Marion. Behavior abstraction in malware analysis. In *Runtime Verification.* 2010.

[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13, 1970.

[5] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 2007. release October, 12th 2007.

[6] P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero. Identifying dormant functionality in malware programs. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[7] N. En and N. Srensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing.* 2004.

[8] M. Fredrikson, M. Christodorescu, and S. Jha. Dynamic behavior matching: A complexity analysis and new approximation algorithms. Technical report, 2011. http://www.cs.wisc.edu/∼mfredrik/matching-tr.pdf.

[9] M. Fredrikson, M. Christodorescu, S. Jha, R. Sailer, and X. Yang. Synthesizing near-optimal specifications of malicious behavior. In *Proceedings of the IEEE Symposium of Security and Privacy*, 2010.

[10] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Adison-Wesley Publishing Company, Reading, Massachusets, USA, 1979.

[11] G. Jacob, H. Debar, and E. Filiol. Malware behavioral detection by attribute-automata using abstraction from platform and language. In *Recent Advances in Intrusion Detection.* 2009.

[12] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. K. andXiaoyong Zhou, and X. Wang. Efficient and effective malware detection at the end host. In *Proceedings of the Usenix Security Symposium*, 2009.

[13] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. Mitchell. A Layered Architecture for Detecting Malicious Behaviors. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, 2008.

[14] F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Transactions on Computational Logic*, 5, 2004.

[15] Y. Park, D. Reeves, V. Mulukutla, and B. Sundaravel. Fast malware classification by automated behavioral graph matching. In *Proceedings of the Workshop on Cyber Security and Information Intelligence Research*, 2010.

[16] Y.-J. Park, Z. Zhang, and S. Chen. Run-time detection of malware via dynamic control-flow inspection. In *Proceedings of the IEEE Conference on Application-specific Systems, Architectures and Processors*, 2009.

[17] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *Proceedings of the Usenix Security Symposium*, 1999.

[18] A. G. Tokhtabayev, V. A. Skormin, and A. M. Dolgikh. Expressive, efficient and obfuscation resilient behavior-based ids. In *Proceedings of the European Symposium on Research in Computer Security*, 2010.

[19] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Behavior based software theft detection. In *Proceedings of the ACM conference on Computer and communications security*, CCS '09, 2009.

[20] T. Werth, M. Wrlein, A. Dreweke, I. Fischer, and M. Philippsen. Dag mining for code compaction. In L. Cao, P. S. Yu, C. Zhang, and H. Zhang, editors, *Data Mining for Business Applications*. 2009.

[21] C. Zhao, J. Kong, and K. Zhang. Program behavior discovery and verification: A graph grammar approach. *IEEE Transactions on Software Engineering*, 36, 2010.