Master Thesis Computing Science: Software Science

Fingerprinting Product Families

Author:
Gianni Monteban

Supervisor:
Daniel Strüber

Second reader:
Frits Vaandrager

Daily supervisor:
Carlos Diego N. Damasceno

Radboud University Nijmegen
Institute for Computing and Information Sciences
Faculty of Science

September 6, 2023

Abstract

During the evolution of a system, various vulnerabilities are often found that should be patched in newer versions. Since not every system is regularly updated, systems often remain vulnerable to attacks. Therefore, it is highly desirable to have an efficient technique to detect if the system is running in a vulnerable version. Software fingerprinting can automatically detect which version is running in a system by observing the behaviour of the system under special test inputs, such as adaptive distinguishing sequences. Due to the size of the signatures, more efficient fingerprinting techniques are required, as fingerprint matching has exponential worst-case complexity. This worst-case complexity of fingerprint matching becomes even larger when we consider a variability-aware system. In such a system, a different selection of features can generate an exponentially large number of products. Current fingerprinting approaches rely on signatures that identify each version independently. By lifting these signatures to one family-based representation, we can drastically reduce the size of signatures and potentially exploit behavioural commonalities and variability to speed up the fingerprinting process. In this work, we present a fingerprinting framework that uses a feature finite state machine to specify and unify the behaviour of multiple versions of a system. We introduce one passive fingerprinting approach and three active fingerprinting approaches: family-based Online Machine Separation, Configuration-based Preset Distinguishing Sequence (CPDS) and Configuration-based Adaptive Distinguishing Sequence (CADS). To analyse the efficiency of the active approaches, we compare them with each other and with a method from the literature. When we consider the efficiency of the calculated sequences, the CPDS and CADS can compute smaller sequences with fewer resets. So with these two approaches, we require less effort to detect the version of a black-box system, which makes it possible to detect the version faster. When we consider the efficiency of computing the sequences, we find that the family-based Online Machine Separation approach is close to the efficiency of the approach from the literature. The other proposed approaches were much slower in calculating the sequences than the method from the literature. However, the CADS could still finish within a reasonable time, while the CPDS sometimes exceed our set time limit.

Contents

Chapter 1

Introduction

In modern systems, free and open source components are typically used to increase productivity and reduce development costs. However, serious security flaws can exist in these components. Sometimes, security flaws exists in a particular configuration of such a component, which arise by activating or deactivating particular features. Such a component is known as a product family and is described by a Software Product Line (SPL). Verifying if all the products of an SPL are secure is very hard, as it is possible to generate 2^n different products by only having n features. Kenner et al. [16] held a survey to investigate techniques for ensuring the security of program families and categorized them into two categories, product-based and family-based analysis. With product-based analysis, one specific configuration of a family is analysed at a time while with family-based analysis, we operate on the domain artefacts and have knowledge about different configurations. With a family-based analysis, we can verify the different products all at once, which can reduce the costs of generating verifying sequences. While there are multiple publications for product-based security analysis, there is limited work on the family-based analysis, with Peldszus et al. [23] being the only publication reported in the study.

Software fingerprinting is a classification of (automated) techniques which can detect if a vulnerable version of a product family is running in a black-box system by using signatures of known systems. These signatures can vary from input/output sequences, the source code or different graph representations. Shu and Lee [24] proposed different methods to fingerprint black-box systems. Two of these methods are categorized as group matching. With group matching, we use a set of signatures to detect which version runs in the black-box system and can be performed in either an active or passive method. With the active method (Online Machine Separation), we interact with a System Under Test (SUT) by applying inputs and observing outputs. With the passive method, a set of in- and output combinations from executions of the SUT is provided and used to determine the version. In terms of their effectiveness of fingerprinting a system, an active method is considered more effective as it has the advantage of selecting an input which separates different versions and can thus detect the version faster. However, the passive method has the advantage that the black-box system is unaware of being fingerprinted as it is operable while we observe the in- and outputs from the system. But this also limits the passive method, since if we have not observed enough in- and outputs from the system, it can be the case that we are unsuccessful in detecting a single version and thus can not fingerprint the black-box system. For active software fingerprinting, the limitation is known to be the computation efficiency [2]. In Shu and Lee, the Online Machine Separation method requires calculating $\mathcal{O}(k^2)$ distinguishing sequences for k signatures [24]. To reduce this cost, Janssen [15] proposed a new active fingerprinting framework based on state distinguishing sequences. But even with this newly proposed method, further research is required to see how the methods behave when more models are added [15].

The current discussed fingerprinting frameworks all use a product-based approach. Damasceno and Strüber [7] present the vision for a family-based approach, specifically, a two-step fingerprinting framework which consists of *fingerprint discovery* and *fingerprint matching*. Compared to existing techniques, their framework [7] is intended to rely on a family-based representation where

the different feature representations are annotated to describe the behaviour of a system. While there are techniques known to support the creation of family models [8] that cover the *fingerprint discovery* step, a technical solution for the *fingerprint matching* step has not been proposed yet.

While there are multiple solutions [24, 15] discussing a product-based (i.e. a signature for every product) approach for software fingerprinting, there is currently no complete solution for a family-based (i.e. one signature describing different products) approach. With a family-based approach, we know the variability and commonalities between the products which can potentially speed up the process of generating fingerprinting sequences. Since we know that efficiency is a problem for software fingerprinting [2], we want to see if we can improve the fingerprinting process by switching to a family-based approach. Therefore, we propose a solution for the *fingerprint matching* step of Damasceno and Strüber [7] by creating a family-based fingerprinter. In this fingerprinter, we use a family-based signature which is the extension of a Mealy machine. As Mealy machines are widely used in model-based testing [19] and fingerprinting [24, 15], we can depend on known techniques to create an efficient family-based fingerprinter. Thus, for this thesis, we formulate the following research goal (RG):

RG: Present an efficient family-based fingerprinter
by lifting the notion of known product-based fingerprinting techniques
and model-based testing techniques to our family-based signature.

For our family-based active method, we propose a total of three new approaches. First, we propose a family-based version of Online Machine Separation by lifting the notion of the regular (product-based) Online Machine Separation proposed by Shu and Lee [24] to product families. In the product-based approach, Shu and Lee pairwise separate a set of PEFSMs signatures, which is an extension of a Mealy machine. By lifting this notion to our family-based representation, we show that this known working technique can also work with our representation and potentially speed up the search for the sequences. Also, this technique can be used as a baseline for the active family-based fingerprinters since it considers $\mathcal{O}(k^2)$ sequences for k products which is not optimal. To find better fingerprinters, we essentially want fewer sequences which can separate all products. To pairwise separate the models, the Online Machine Separation method depends on the theory of state identification. With state identification, we determine the initial state of a Mealy machine based on an input/output experiment [19]. These experiments are known as state distinguishing sequences and are classified in either a *preset* or *adaptive* distinguishing sequence. Here a preset distinguishing sequence (PDS) is a single sequence while the adaptive distinguishing sequence (ADS) is a tree where the input depends on the seen output. However, not every Mealy machine does have a distinguishing sequence. Since the ADS is a tree where the inputs depend on the outputs, this sequence exists more often in a Mealy machine and is often shorter than the PDS. As the adaptive and preset method can separate all the states in a Mealy machine using a *single* sequence, we can potentially improve the length of the sequence by lifting these methods to the notion of our family-based representation where we separate the products of an SPL. Therefore we propose a Configuration-based PDS and a Configuration-based ADS where the configuration is the representation of a product of an SPL. Now, to overcome the limitation that not every Mealy machine has a distinguishing sequence, we assume that every system is resettable to its initial state. In this way, it will always be possible to separate products which do not have equivalent behaviour.

To verify if we propose efficient active fingerprinting methods, we analyse our proposed family-based approaches and product-based Online Machine Separation approach in terms of their efficiency. For efficiency, we follow a typical evaluation setup from the model-based testing domain [1], which considers two aspects of efficiency. At first, the efficiency of generating the sequence in terms of execution time. Ideally, an approach has a low execution time to find the sequences. Second, the efficiency of the sequence in terms of their size, which is the number of inputs and system resets in our case. Ideally, a sequence has a low number of inputs and system resets, when for example fingerprinting a TLS server implementation, as done by Janssen [15], applying an input causes extra overhead such as network latency and cryptography operations. When considering a reset, the client needs to close the current connection and reconnect to the server, which is also a time-consuming operation.

In the remainder of this paper, we first discuss the background material related to the family model. In Chapter 3, we discuss work related to the topic of this thesis. Chapter 4 introduces different ways of performing fingerprinting using the family model. In Chapter 5, we compare the proposed methods of fingerprinting against each other and to a method we found in the literature. Finally, in Chapter 6, we draw the conclusion of the thesis.

Chapter 2

Background

In this section, we revisit the background of this thesis: Finite State Machines (FSMs), State identification, Software Product Lines (SPLs), Featured Finite State Machine (FFSM), FFSM$_{diff}$ algorithm, and the basis of software fingerprinting.

2.1 Finite State Machine (FSM)

A Finite State Machine (FSM) consists of states and directed edges [13]. Formally, an FSM is defined as 5-tuple (S, s_0, I, O, T). In Definition 2.1 the complete formal definition of an FSM can be found.

Definition 2.1 *An FSM M is a 5-tuple (S, s_0, I, O, T):*

1. *S is a finite set of states*
2. *s_0 is the initial state*
3. *I is a finite set of inputs*
4. *O is a finite set of outputs*
5. *T is a set of transitions. Here $t = (s, x, o, s') \in T$, where $s \in S$ is the source state, $x \in I$ is the input label, $o \in O$ is the output label, and $s' \in S$ is the target state.*

An FSM is defined to take an edge under a specified input. With this edge, we move to a state and produce a specified output. This behaviour is specified using a state transition function and output function. With the state transition function, we take a state S and an input I and reach a new state S, i.e. $\delta : S \times I \to S$. With the output function, we take a state S and an input I and observe the corresponding output O, i.e. $\lambda : S \times I \to O$. By definition, these function can also handle a block of states in the following manner [19]:

$$\delta(B, x) = \{\delta(s, x) | s \in B\}$$

Next to the formal definition, it is possible to visualize an FSM as a directed graph. An example can be seen in Figure 2.1.

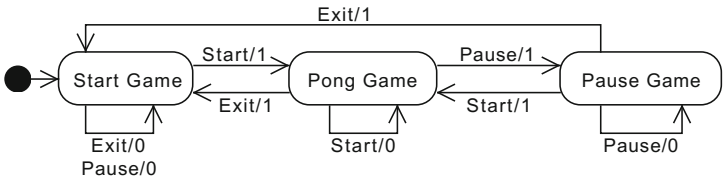


Figure 2.1: Finite State Machine Pong Game [14]

To be able to generate test- or distinguishing sequences, certain properties need to hold for an FSM. These properties are called *validation properties*. In Definition 2.2, we have listed the important ones for this study.

Definition 2.2 ***FSM Validation Properties***

Note: $\bullet$ *is used to separate the different clauses to make it more readable.*

- ***Deterministic****: If two transitions leave a state with a common input, then both transitions reach the same state while producing the same output, i.e.,* $\forall(s, x, o, s'), (s, x, o', s'') \in T \bullet s' = s'' \wedge o = o'$.
- ***Complete****: Every state has one transition for each input, i.e.,* $\forall s \in S, x \in I \bullet \exists o \in O, s' \in S \bullet (s, x, o, s') \in T$.

2.1.1 State identification

In this paper, we approach the task of fingerprinting using strategies for state identification. With these strategies, we want to determine the initial state of a Mealy machine by knowing the states and behaviour of a machine. This can be done by applying inputs and observing outputs from the system. Such an experiment is called a *distinguishing sequence*. Typically, these are classified as preset- or adaptive distinguishing sequences.

2.1.1.1 Preset Distinguishing Sequence (PDS)

A Preset Distinguishing Sequence (PDS) is a single input sequence that can distinguish the states of a system, i.e. $\langle\langle Pause\rangle, \langle Exit\rangle\rangle$ for Figure 2.1. To calculate this sequence, we need to define two *uncertainties*. With the uncertainty, we want to keep track of a split after an input sequence (*initial state uncertainty*) or keep track of the state which we reached after the input sequence (*current state uncertainty*) [20]. Formally, the concepts are defined for an FSM $M = (S, s_0, I, O, T)$ and x being the input sequence of it in the following manner:

- **Initial uncertainty**: $\pi(x)$ is the partition $\{C_1, C_2, \ldots, C_r\}$ of S such that s and $s' \in S$ are in the same block C_i if and only if $\lambda(s, x) = \lambda(s', x)$.[19]
- **Current uncertainty**: $\sigma(x) = \{\delta(C, x) | C \in \pi(x)\}$.[19]

If we would consider the FSM of Figure 2.1 and input trace $Exit$, the uncertainties would look the following:

$$\pi(Exit) = \{\{Start\_Game\}, \{Pong\_Game, Pause\_Game\}\}$$
$$\sigma(Exit) = \{\{Start\_Game\}, \{Start\_Game, Start\_Game\}\}$$

A typical approach to find the PDS is by using a *successor tree*. With this tree, we start at the root with all of the states of the FSM. Then for every input sequence, the tree contains a path starting from the root to a corresponding node. In every node, we annotate the corresponding current and/or initial state uncertainty. In Figure 2.2 we see an example of such a tree where $\langle\langle Pause\rangle, \langle Exit\rangle\rangle$ is the PDS as *Start*, *Pause* and *Pong* are divided into a singleton set.

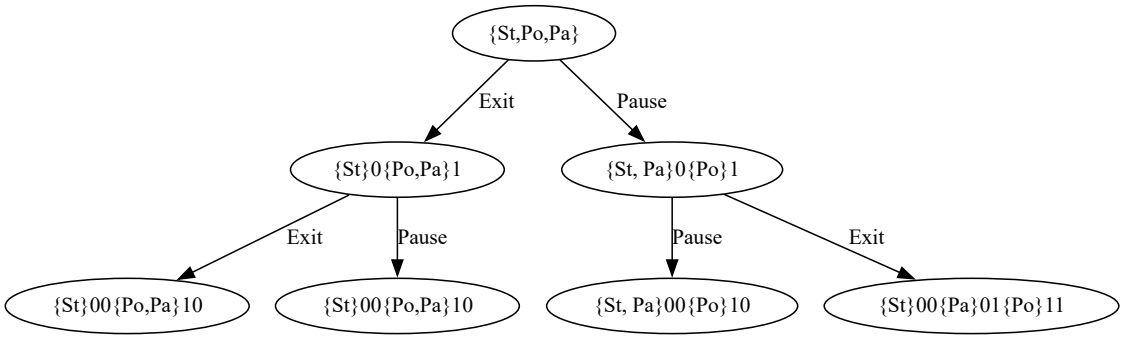


Figure 2.2: Example successor tree for Figure 2.1 with inputs = {Exit, Pause}

Another structure to find a PDS is the *super graph*.

Definition 2.3 *The* ***super graph*** *of an FSM $\mathcal{M}$ is a directed graph G which is denoted by Super-G($\mathcal{M}$).*

- *The set of vertices of Super-G($\mathcal{M}$) is Super($\mathcal{M}$),*
- *The initial vertex of Super-G($\mathcal{M}$) is $V_0 = \{S\}$,*
- *The edges of Super-G($\mathcal{M}$) are labeled by the input symbols of $\mathcal{M}$,*
- *For an input symbol a and $V, V' \in Super(\mathcal{M}) : V \rightarrow^a V'$ if and only if a is valid for V and $V' = \mu(V, a)$.*

Here Super($\mathcal{M}$) and μ are defined as followed [19]:

- *Super($\mathcal{M}$) is the set of multi-sets of non-empty blocks of states such that $W = \{B_1, B_2, \ldots, B_l\}$ is in Super($\mathcal{M}$) if and only if $\sum B_i \in W |B_i| = |S|$. Example for $S = \{s_1, s_2\}$:*

$$Super(\mathcal{M}) = [\{\{s1\}, \{s2\}\}; \{\{s1\}, \{s1\}\}; \{\{s2\}, \{s2\}\}; \{\{s1, s2\}\}]$$

- *$\mu(\tau, a)$ is the output function for each block of τ for a given input a. By partitioning each member C of τ with respect to $\lambda(\cdot, a)$, then applying $\delta(\cdot, a)$ on each block of the obtained partition. For example on Figure 2.1:*

$$\mu(\{\{Start\_Game, Pong\_Game, Pause\_Game\}, Exit\}) =$$

$$\{\{Start\_Game\}_0, \{Pong\_Game, Pause\_Game\}_1\}$$

The main difference between the super graph method and the successor tree is the fact that the super graph is a finite structure. With the successor tree, every sequence which can be made is decorated in the tree, which leads to an infinite structure. Also, the super graph only considers *valid* inputs while the successor tree contains all inputs. An input is considered *valid* if on an input a the states s and s' either produce distinct output symbols or move to distinct states. In Figure 2.3 an example is given of the super graph where $\langle\langle Pause\rangle, \langle Exit\rangle\rangle$ is the PDS as all the states are separated in singleton sets.

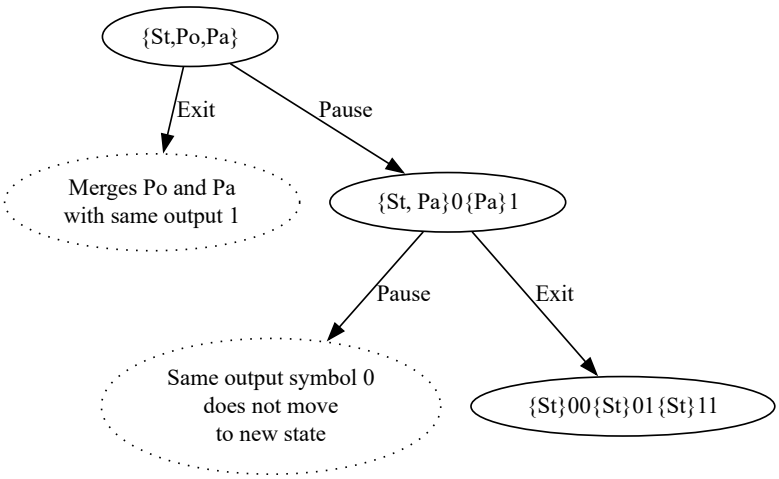


Figure 2.3: Example of super graph for Figure 2.1 with inputs = {Exit, Pause}

2.1.1.2 Adaptive Distinguishing Sequence (ADS)

Where a PDS can be shown as a simple sequence, this is not possible for an ADS. With an ADS, we have an experiment where the next input is dependent on the seen output. This makes the ADS more powerful than the PDS as it exists more often and can be shorter. In Definition 2.4 the formal definition is given for an ADS and in Figure 2.4b we see the example of an ADS for the given FSM in Figure 2.1.

Definition 2.4 *An ADS [19] of a Mealy machine $M = (S, s_0, I, O, T)$ is a rooted tree T such that:*

- *The number of leaves of T equals the cardinality of S*
- *The leaves of T are labeled with states of M*
- *The internal nodes of T are labeled with input symbols from I*
- *The edges of T are labeled with output symbols from O*
- *Edges leaving from a common node are labeled with distinct output symbols*
- *For each leaf u of T, if S_u is the label of u, x_u and y_u are respectively the input and output sequences formed by the concatenation of the node and edge labels on the path from the root to the leaf u, then $\lambda(s_u, x_u) = y_u$*
- *The length of an ADS is defined by the depth of the tree T*

Lee and Yannakakis [21] propose a method for computing an ADS with polynomial size in polynomial time. To reach this, first an intermediary structure is introduced: a splitting tree. In Figure 2.4a an example is given and the structure is defined as follows:

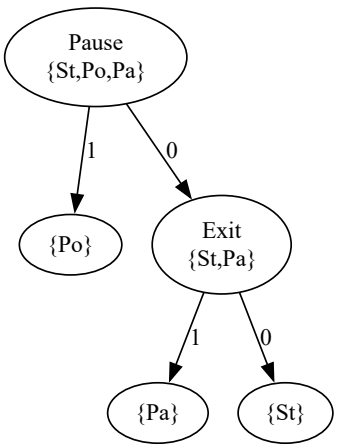
Definition 2.5 *A **splitting tree** [19, 21] associated with a FSM $M = (S, s_0, I, O, T)$ is a rooted tree G such that:*

- *Each node of G is labeled by a subset of S and the root of G is labeled with S*
- *The children's labels of an internal node of G are disjoint and the label of an internal node of G is the union of its children's labels*
- *With each internal node of G is associated an input sequence*
- *With each internal node u of G, with associated set-label S_u and input string label x_u, is associated a mapping $f_u : S_u \to S$ such that $f_u(s) = \delta(s, x_u)$ for each s in S_u*
- *Each edge of G is labeled with an output symbol from O*

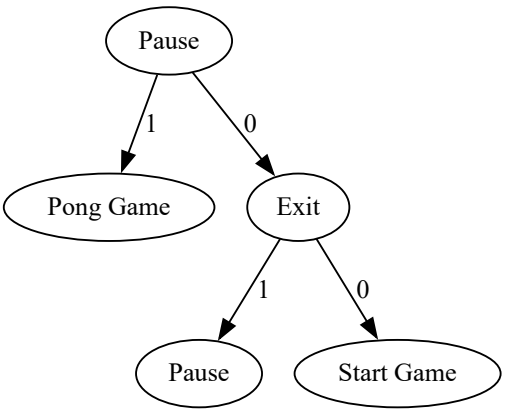
*Let $\pi(G)$ denotes the collection of sets of states formed by the labels of the leaves of the splitting tree G. G is a **complete splitting tree** if $\pi(G)$ is the discrete partition of S.*

To build this splitting tree, the algorithms considers three types of valid inputs. Note that we first try to apply all a-valid inputs on all the leaves before continuing with b-valid inputs.

1. a-valid input: for a set of B states there exists an input a which refines to multiple blocks.
2. b-valid input: for a set of B states there exists an input a for which there exists a node in G where the set-label contains $\delta(B, a)$. If σ is the input associated with the found node, we know that B can be refined by $a\sigma$.
3. c-valid input: move the set of B states to another set of C states which already have been refined with input τ. If σ is the input associated with the C states node, we know that B can be refined by $\tau\sigma$.



(a) Splitting tree example for Figure 2.1



(b) ADS example for Figure 2.1

Figure 2.4

2.2 Software Product Line (SPL)

A Software Product Line describes a family of software systems by using a common and managed set of features [8]. By selecting a set of features, it is possible to define different products. The relation between the features can be represented in a feature diagram, as shown in Figure 2.5. In a feature diagram, different conventions are used to represent the commonalities and variability of an SPL. The commonalities are represented by a *mandatory* option while for the variability there are different representations possible. At first, we have the *optional* selection, this feature can be selected but it is not necessary. Next to this, we have an *or* selection. Now at least one of the features must be selected. At last, we have the *alternative* selection. Here only one of the options is allowed to be picked. Whenever a feature is selected, its parents must also be selected. In Table 2.1, all the possible products together with the constraint itself are listed.

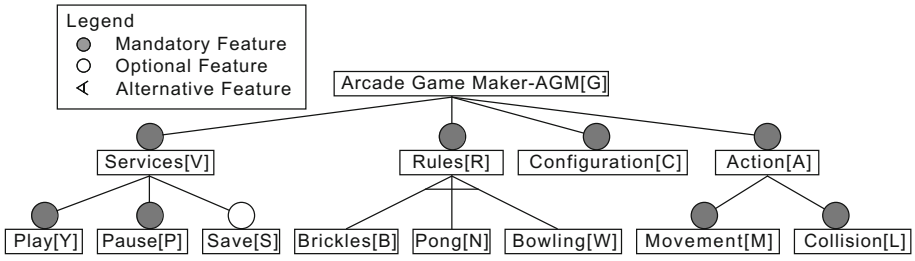


Figure 2.5: Feature model Arcade Game Machine [14]
Note: Features here have a shorthand letter, which is not the standard syntax

Product no	Feature selection	Description
1	$\{G, V, R, C, A, Y, P, S, B, M, L\}$	Brickles and save
2	$\{G, V, R, C, A, Y, P, B, M, L\}$	Brickles without save
3	$\{G, V, R, C, A, Y, P, S, N, M, L\}$	Pong and save
4	$\{G, V, R, C, A, Y, P, N, M, L\}$	Pong without save
5	$\{G, V, R, C, A, Y, P, S, W, M, L\}$	Bowling and save
6	$\{G, V, R, C, A, Y, P, W, M, L\}$	Bowling without save

Table 2.1: Features possible for AGM feature model

2.2.1 Featured Finite State Machine (FFSM)

For describing the behaviour of a single system, the Mealy machine (FSM, [13]) representation is a widely used representation. When we consider an SPL which can produce different product systems from one code base, an FSM can only model the generated products. By extending the FSM to a Feature Finite State Machine (FFSM, [14]), it becomes possible to encode the different features inside an FSM. In this way, we have a single variability-aware representation which describes the behaviour of a whole SPL. To create an FFSM from an FSM, we need to extend the states and transitions. By annotating these with feature constraints from the SPL, we gain knowledge on which state or transitions is seen with which feature. Formally, an FFSM is defined as a 7-tuple: $(F, \Lambda, C, c_0, Y, O, \Gamma)$. In Definition 2.6 the complete formal definition of an FFSM can be found.

Definition 2.6 *an FFSM is a 7-tuple:* $(F, \Lambda, C, c_0, Y, O, \Gamma)$.

1. F *is a finite set of features*
2. Λ *is the set of product configurations*
3. $C \subseteq S \times B(F)$ *is a finite set of* ***conditional states****, where* S *is a finite set of state labels,* $B(F)$ *is the set of all feature constraints, and* C *satisfies the following condition:*

$$\forall_{(s,\varphi)\in C} \bullet \exists_{\rho\in\Lambda} \bullet \rho \models \varphi$$

4. $c_0 = (s_0, true) \in C$ *is the* ***initial conditional state***
5. $Y \subseteq I \times B(F)$ *is a finite set of* ***conditional inputs****, where* I *is the set of input labels*
6. O *is a finite set of* ***outputs***
7. $\Gamma \subseteq C \times Y \times O \times C$ *is the set of conditional transitions satisfying the following condition:*

$$\forall_{((s,\varphi),(x,\varphi''),o,(s',\varphi'))} \in \Gamma \bullet \exists_{\rho\in\Lambda} \bullet \rho \models (\varphi \wedge \varphi' \wedge \varphi'')$$

An FFSM has also some more similarities to an FSM. For an FSM we have different *validation properties*. These properties can also be lifted to the FFSM definition. These properties are listed in Definition 2.7. FFSMs can be visualized by using the graph representation of FSMs, extended by the feature constraints in the edges and states. In Figure 2.6, we see that some states and transitions are annotated with feature constraints while others are not. If a state or transition is annotated, it is only seen if the specific feature holds, if it is not annotated, the behaviour is always seen.

Definition 2.7 ***FFSM Validation Properties***

- ***Deterministic****: an FFSM is* ***deterministic*** *if for all conditional states when there are existing two enabled conditional transitions with the same input for a product* ρ*, then both transitions lead to the same state, i.e.,*

$$\forall (s,\varphi) \rightarrow_o^{x,\varphi'} (s',\varphi_a), (s,\varphi) \rightarrow_o^{x,\varphi''} (s'',\varphi_b) \in \Gamma \bullet \forall_{\rho\in\Lambda} \bullet \rho \not\models (\varphi \wedge \varphi' \wedge \varphi'' \wedge \varphi_a \wedge \varphi_b) \vee s' = s''$$

- ***Complete**: an FFSM is **complete** if for all conditional states in a product ρ there is an outgoing valid transition for each and every input, i.e.,*

$$\forall_{(s,\varphi)\in C} \bullet \forall_{\rho\in\Lambda} \bullet \forall_{x\in I} \bullet \rho \not\models \varphi \vee \exists_{(s,\varphi)\to_o^{(x,\varphi'')}(s',\varphi')\in\Gamma} \bullet \rho \models \varphi' \wedge \varphi''$$

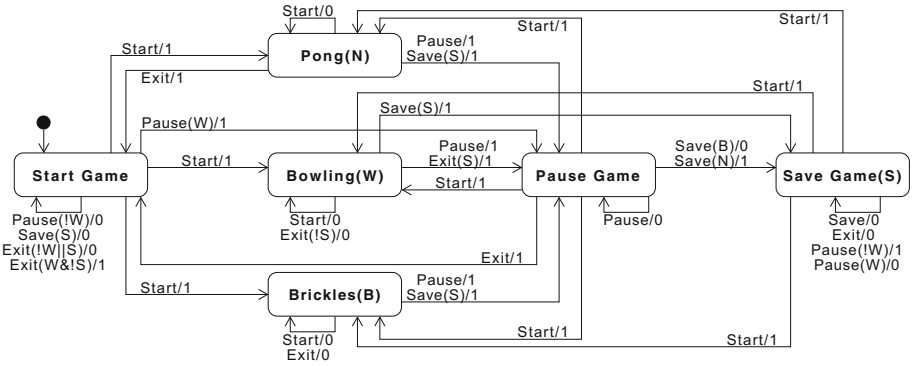


Figure 2.6: FFSM [14]

2.2.1.1 FFSM$_{diff}$

As the current family systems only have an FSM representation available of the products, we need to have a method to create an FFSM. The FFSM$_{diff}$ algorithm [8] is an adaptation of the LTS$_{diff}$ algorithm [27]. With this algorithm, it is possible to match and merge the states of an FSM and annotate the features to create an FFSM model. In Algorithm 2.1 the complete pseudo code is shown.

Algorithm 2.1 FFSM$_{diff}$ algorithm

```
1  Input: Model M_r, Model M_u, k, t, r;
2  PairsToScore = computeScores(M_r, M_u, k);
3  KPairs = identifyLandmarks(PairsToScore, t, r);
4  NPairs = ∪_{(a,b)∈KPairs} Surr(a,b) − KPairs;
5  while NPairs ≠ ∅ do
6      while NPairs ≠ ∅ do
7          (a,b) = pickHighest(NPairs, PairsToScore);
8          KPairs = KPairs ∪ (a,b);
9          NPairs = removeConflicts(NPairs, (a,b));
10     end
11     NPairs = ∪_{(a,b)∈KPairs} Surr(a,b) − KPairs;
12 end
13 Add = {b1 -a/b-> b2 ∈ D_u | ∄(a1 -a/b-> a2 ∈ D_r ∧ (a1,b1) ∈ KPairs ∧ (a2,b2) ∈ KPairs)};
14 Rem = {a1 -a/b-> a2 ∈ D_r | ∄(b1 -a/b-> b2 ∈ D_u ∧ (a1,b1) ∈ KPairs ∧ (a2,b2) ∈ KPairs)};
15 Kpt = KPairs;
16 FF_{r,u} = mergeAndAnnotate(M_u, M_r, Add, Rem, Kpt);
17 return (FF_{r,u});
```

In the *computeScores* function, the matching score for all the states is calculated by solving a set of linear equations. After that, in the *identifyLandmarks* function, the pairs which are at least in the top t% and are at least r times as good as the other options are added as a match. By default also the initial states (s_{0_r}, s_{0_u}) is added as a match. After that, the highest surrounding

pair is added to the set of matches. This process is repeated until there are no more surrounding pairs. After the matches are calculated, the added and removed transitions are calculated. These are used in the `mergeAndAnnotate` function where the two FSMs are merged into one single FFSM. In Definition 2.8 the function is formally defined.

Definition 2.8 *As noted in Definition 2.6, an FFSM is a 7-tuple* $FF = (F, \Lambda, C, c_0, Y, O, \Gamma)$. *This 7-tuple can be created from two FSMs after the* LTS_{diff} *algorithm. Let's consider the machines* $M_r = (S_r, s_{0_r}, I_r, O_r, T_r)$ *and* $M_u = (S_u, s_{0_u}, I_u, O_u, T_u)$. *Here* p_r *and* p_u *are the products and* ξ_u *and* ξ_r *are the configurations for the FSMs.*

- $F = (p_r \cup p_u)$ *is the set of features implemented by the two products*
- $\Lambda = \{\xi_r, \xi_u\}$ *is the smallest set composed by the two configurations*
- $C \subseteq (S_r \cup S_u \cup (S_r \times S_u)) \times B(F)$ *is the set of conditional states where*

$$\forall s_i \in S_r, s_j \in S_u | (s_i, s_j) \in KPairs \cdot ((s_i, s_j), \xi_r | \xi_u) \in C,$$

$$\forall s_i \in S_r, \nexists s_j \in S_u | (s_i, s_j) \in KPairs \cdot (s_i, \xi_r) \in C,$$

$$\forall s_j \in S_u, \nexists s_i \in S_r | (s_i, s_j) \in KPairs \cdot (s_j, \xi_u) \in C,$$

- $c_0 = ((s_{0_r}, s_{0_u}), true) \in C$ *is the initial conditional state*
- $O = (O_r \cup O_u)$ *is the finite set of output symbols*
- $\Gamma \subseteq C \times Y \times O \times C$ *is the set of conditional transitions where*
  - *two transitions* $(s_i, x, o, s_i') \in T_r$ *and* $(s_j, x, o, s_j') \in T_u$ *are unified in the same conditional transition if their source and destination states are merged, i.e.,* $(s_i, s_j), (s_i', s_j') \in KPairs$. *Formally, this means that:*

$$\forall (s_i, x, o, s_i') \in T_r, (s_j, x, o, s_j') \in T_u | (s_i, s_j), (s_i', s_j') \in KPairs$$

$$(((s_i, s_j), \phi), (x, (\xi_u | \xi_r)), o, ((s_i', s_j', \phi'))) \in \Gamma$$

  - *otherwise, for two transitions* $(s_i, x_i, o_i, s_i') \in T_r$ *and* $(s_j, x_j, o_j, s_j') \in T_u$, *there are two independent conditional transitions defined as follows:*

$$((s_i, \phi_r), (x, \xi_r), o_i, (s_i', \phi_r'')) \in \Gamma$$

$$((s_j, \phi_u), (x, \xi_u), o_j, (s_j', \phi_u'')) \in \Gamma$$

2.3 Software fingerprinting

Software fingerprinting is a class of (automated) techniques to recognize a system based on a set of signatures. Shu and Lee [24] addressed the task of fingerprinting an extended FSM, namely a Parameterized Extended Finite State Machine (PEFSM). This PEFSM is used as the behavioural signature of a system under test. Compared to a standard FSM, the PEFSM has parameterized input and output symbols and has variables as part of the states and transitions. With this signature, Shu and Lee discuss a range of tasks involved in the fingerprinting domain. In Table 2.2, the different tasks are listed. In this paper, we focus on the task of group matching.

Fingerprint type	**Active Experiment**	**Passive Experiment**
Matching	PEFSM Conformance testing	PEFSM Passive testing
Group Matching	Online Machine Separation	Concurrent passive testing
Discovery with Spec.	Machine enumeration and separation	Back-tracing based passive testing
Discovery without Spec.	FSM supervised learning	*No efficient solution*

Table 2.2: Fingerprint matching [24]

With group matching, the goal is to detect the version of a black-box system by using a set (group) of candidates. This is done by detecting which behaviour of the candidates matches with the system. For this, there are existing two ways of group matching, an passive or active way.

In a passive experiment, an input/output trace from the black-box system is observed. This trace is given to the fingerprinting system which returns the set of possible candidates. This has two benefits, at first, the black-box is operable as normal and second, the system is unaware that is being fingerprinted. However there is also a drawback, sometimes we do not have observed enough traces from the system, in this case, we can not fingerprint a single system but end up with a set of potential fingerprints.

In an active experiment, input sequences are calculated and then executed on the black-box system. Based on the output of the system, it can be possible to determine a candidate as the version. This has the benefit that we can always detect the version of the system. However, since we are interacting with the black-box system, the system knows it is being fingerprinted.

For the passive fingerprinting, Shu and Lee consider a set of PEFSMs as candidate group C and a trace T from some implementation. The goal is to check if a machine can generate trace T. This is done by testing all the models concurrently. The first step is to calculate a reachability graph for every PEFSMs (line 6). This is essentially an FSM representation of the PEFSM. After that, the algorithm considers every model as a match, and for every model, all the states are considered as an *uncertainty* (line 7). Then for every step in the trace, we remove the states that do not match the trace from the *uncertainty* and remove the model if *uncertainty* is empty (line 13 - line 22). If we eventually find one model or traversed the whole trace, we stop the process and return the model candidates. In Algorithm 2.2, the pseudocode for the passive group matching is presented. While the passive experiment only works for PEFSMs and does not work with our family-based model. There are some ways to adapt it and use it with our family-based model. An extension of this method is discussed in Chapter 4.2.

Algorithm 2.2 Passive Group Matching - PEFSMs

```
1:  Input: candidate group C = {M_1, M_2, ...., M_k}
2:  trace T = ⟨⟨I_0, O_0⟩, ⟨I_1, O_1⟩, ..., ⟨I_{L-1}, O_{L-1}⟩⟩
3:  Output: possible candidate set PC;
4:  PC = C
5:  for all i in [1..k] do
6:      Calculate minimized reachability graph G_i;
7:      uncertainty[i] = {G_i.states};
8:  end for
9:  id = 0;
10: while PC.size > 1 and id < L do
11:     for all M_i in PC do
12:         new_uncertainty = {}
13:         for all t⟨S_src, S_dst, I_t, O_t⟩ in G_i do
14:             if (S_src ∈ uncertainty[i] and I_id == I_t and O_id == O_t) then
15:                 new_uncertainty = new_uncertainty ∪S_dst
16:             end if
17:         end for
18:         if new_uncertainty == {} then
19:             PC = PC - M_i
20:         else
21:             uncertainty[i] = new_uncertainty
22:         end if
23:         id = id +1
24:     end for
25: end while
26: return PC
```

In the proposed active fingerprinting by Shu and Lee, called Online Machine Separation, they also consider a set of PEFSMs as candidate group C. With this set, they propose a method to generate a fingerprinting set F. This set is built by adding sequences that separate different machines from the set C. A sequence can separate two machines of C, if for a given input sequence the machines produce different outputs, i.e., $C = \{M_1, M_2, ..., M_k\}$ a sequence *seq* separates M_i and M_j if $f_{output}(M_i, seq) \neq f_{output}(M_j, seq)$. To generate these separating sequences, we assume that every candidate can be separated from the other. With this given, at first, a set is initialized with all the candidates C. The next step is then to pairwise separate the machines by picking two machines that are not separated yet. Then, a separating sequence is calculated (line 6). This separating sequence is then executed over all the candidates which then are split according to their outputs (lines 11 to 17). This process continues until all machines are separated from each other. The pseudocode for active group matching is given in Algorithm 2.3.

Algorithm 2.3 Online Machine Separation - PEFSMs

```
1:  Input: candidate group C = {M_1, M_2, ...., M_k}
2:  Output: fingerprint set F
3:  F = {}
4:  partition = {{1, 2, ..., k}}
5:  while partition.size < k do
6:      find M_i and M_j in same partition set
7:      calculate separating sequence SEQ for M_i and M_j
8:      for all set S in partition do
9:          undef = {}; remove S from partition
10:         initialize map as an empty map from A* to POW(S)
11:         for all x in S do
12:             if SEQ is defined for M_x then
13:                 map(f_output(M_x, SEQ)) ∪= {x}
14:             else
15:                 undef = undef ∪{x}
16:             end if
17:         end for
18:         for all s* in range(map) do
19:             if s*∪undef ∉ partition then
20:                 partition = partition ∪{s*∪ undef}
21:             end if
22:         end for
23:     end for
24:     F = F ∪ {SEQ}
25: end while
26: return F
```

2.3.1 Family-based fingerprinting

Damasceno and Strüber [7] proposed the development of a framework for family-based fingerprint analysis. The framework is divided into two stages: (a) *Fingerprint discovery*, where the family signature is created and (b) *Fingerprint matching*, where the family signature is used to query the black-box system. In Figure 2.7, the different stages are shown with its intermediary steps. In this thesis, we mostly focus on step C as we want to improve the efficiency to generate the matching sequences and the efficiency to detect the system.

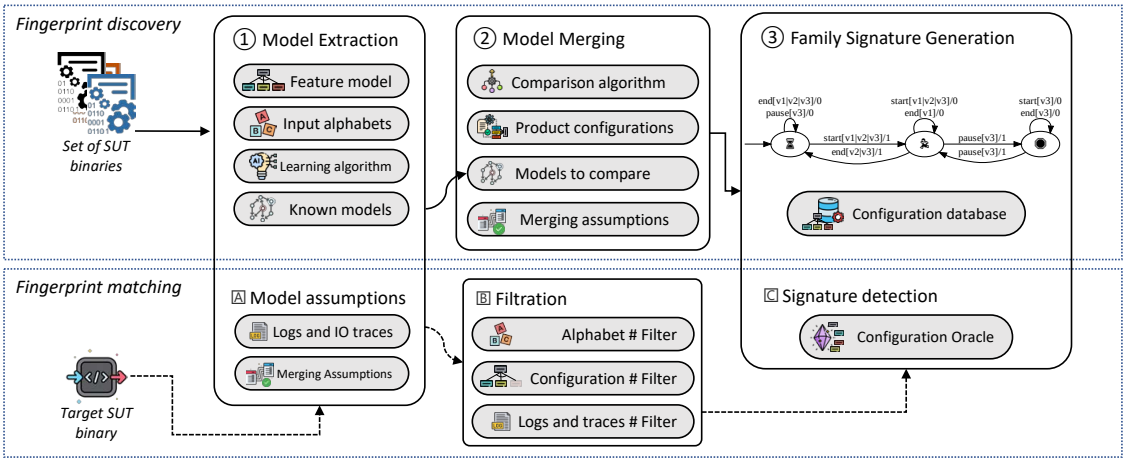


Figure 2.7: Framework for family-based fingerprinting [7]

Chapter 3

Related Work

In this chapter we discuss related work conducted in the topics of fingerprinting and security of software product lines.

3.1 Fingerprinting

Alarabee et al. [2] present a survey on fingerprinting binary files. They describe three categories: *Binary Static Analysis*, *Binary Dynamic Analysis* and *Binary Hybrid Analysis*. With *Binary Static Analysis*, the binary files are analysed without executing them. By extracting signatures in the form of interfaces, dependencies and functionalities one can reverse engineer the binaries to identify them. The challenge however is, as discussed by Kinder et al. [17], that a lot of data is lost when compared to performing this analysis on the source code level. With the *Binary Dynamic Analysis*, the binary files are executed and their behaviour is analysed during the execution. By adding an input/output pairing, the behaviour can be better understood [10]. With dynamic analysis, it is not required to dissemble binary code into assembly code. This makes it potentially easier to detect malware binaries as they are typically packed and obfuscated making it infeasible for the application of static analysis. At last, we have the *Binary Hybrid Analysis*, this option combines static and dynamic analyses, the static analysis is first executed and the obtained information is later used in the dynamic analysis. In this paper, we also fingerprint the black-box system by doing a Binary Dynamic Analysis.

As discussed by Shu and Lee [24] and in Section 2.3, dynamic approaches may rely on active or passive experiments. Next to Shu and Lee, François et al. [12] have created a passive fingerprinter based on the extension of a Finite State Machine, the TR-FSM [11]. This is a tree-structured parameterized finite state machine with time-annotated edges. The machine is created by observing different traces from the system-under-test and decorating this inside the tree. In Figure 3.1, an example of two TR-FSMs is shown. In the nodes, we either find the emitted (prefixed by !) or the received (prefixed by ?) message. At the edges, we find the average time it takes for the machine to send the response. When we have observed the state machines of the different systems, we need a method to compare the machines. For this François et al. introduced a kernel function that considers the similar paths (*sim_paths*) from the root and computes a time-based difference. The formula is given below:

$$K(t_i, t_j) = \sum_{p \in sim\_paths} \sum_{edge \in p} e^{-\alpha |delay(edge,t_1) - delay(edge,t_2)|}$$

In Figure 3.1 we see grey and white nodes, the grey nodes are considered to be equal and the white ones are considered to be not equal. In our representation, the FFSM, it is also possible to annotate the edges with the time. But in this thesis, the focus is on untimed state machines and thus an extension for a timed FFSM model is out of the scope of this thesis.

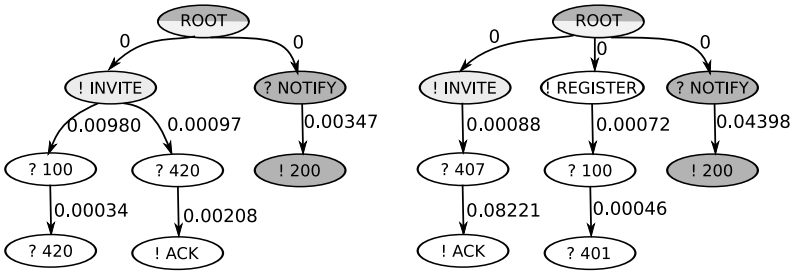


Figure 3.1: TR-FSM example [12]

Janssen [15] discussed two methods to actively fingerprint systems, one which uses an adaptive distinguishing graph (ADG) and another which uses a heuristic decision tree (HDT). Both methods are built from a set of candidate finite state machine models. If the state machines are not available, they are obtained by using model learning on a running system. For the adaptive distinguishing graph [26], the first step is to combine the FSM models into one large FSM. The next step is to convert the large FSM to a labelled transition system (LTS) [25]. This is needed since the ADG algorithm implementation only accepts this representation. After executing the algorithm, we obtain the ADG. An example of an ADG is shown in Figure 3.2. By applying the nodes as an input on the black-box system and following the corresponding output edge, eventually, you end up in a leaf. This leaf represents the version of the black-box system.

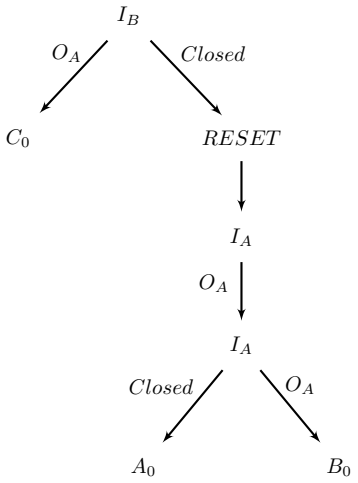


Figure 3.2: ADG example [15]

For the heuristic decision tree, the finite state machines are not combined. First, the models are normalized by converting every state machine into a tree. In these trees, every branch represents a possible path from the original state machine. Next, we combine all newly created trees, this means that if two models share the same branch they end up in the same leaf. This leaf is then annotated with the versions in which the branch exists. The last step is to condensed the tree, this means that all branches that do not add any distinguishing information are be shortened at much as possible. In Figure 3.3 we see an example of a condensed HDT. Now, to identify a version using the HDT, we start at the root of the tree and need to pick an input. We this input to the black-box system and observe the output. In the HDT we follow the corresponding input- and output edge and we either end up in a leaf or node. With a node, we need to again pick an input

until we reach a leaf. When we end up in the leaf, we update the tree with the information we found, i.e. if the whole tree contains version A, B, and C and we find a leaf with A, B we remove C from the tree. Now we reset the black-box system and start at the top of the tree again.

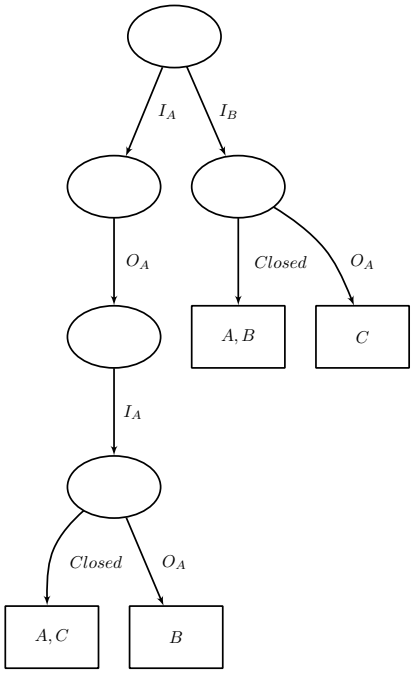


Figure 3.3: HDT example [15]

From these two fingerprinters, Janssen found that it took only a couple of minutes to create the HDT while for the ADG it costs around 10 - 15 minutes. When we consider the efficiency of the trees in terms of inputs and resets, the HDT with a Gini (count) input picker performed better than the ADG. But due to the limited set of models, it is hard to conclude that the HDT Gini (count) performs better in all the cases.

The HDT and ADG approach both use a set of FSMs as the candidates, while in this work we use the FFSM as a unified representation of the candidates. By making use of the FFSM representation, it is potentially also possible to implement the HDT and ADG approach. The ADG approach computes a similar tree to the Configuration-based ADS we propose in this work. Also, the HDT approach normalizes the finite state machines and then combines these trees into the HDT, with the FFSM we already have combined representation and could thus directly generate the HDT. Due to the focus on fingerprinting based on state identification methods, directly generating the HDT is out of the scope of this study.

3.2 Security of Software Product Lines (SPL)

In this work, we want to increase the security of a product derived from an SPL by using a family-based approach. As concerns about the security of an SPL are not new, other papers have discussed the same problem. Kenner et al. [16] present a literature review on the security of SPLs. They evaluated a total of 24 papers that address the security aspects of configurable software systems. Kenner et al. [16] saw a lack of research on mitigating security threats with a family-based approach as there was only one paper published on this topic. By security threats, we mean the vulnerabilities that occur during the life cycle of a system or that may occur when configuring a configurable system. In the only published paper, Peldszus et al. [23] proposed a family-based approach to efficiently execute security checks and obtain the subset of features to return an insecure product. Due to the lack of research, Kenner et al. defined the following literature gap:

investigation of vulnerabilities caused by the configurability of a system and strategies to mitigate such vulnerabilities and avoid them for all variants [16]. Our technique helps to investigate vulnerabilities caused by the configurability of a system. By fingerprinting a derived product using the family-based representation, we detect whether the black-box system is running a vulnerable version. Since we use a family-based representation, we also introduce a new method for validating the security of the system for which there is currently only one proposed solution [23].

Another way to verify if an SPL is conform to the specification can be done by using model checking or testing [6]. These techniques use a model which can handle variability, such as a Featured Transition System (FTS) [4]. In this model, different transitions are annotated with features similar to the presence conditions of an FFSM (Section 2.2.1). An example of an FTS is given in Figure 3.4, $\{f, c, s, t\}$ represent the features of the system.

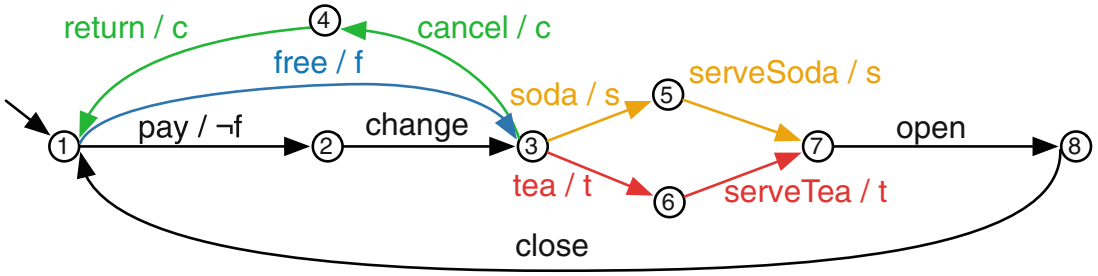


Figure 3.4: Feature Transition System example [6]

Chapter 4

Family-based fingerprinting

With family-based fingerprinting, we consider a modelling approach that can describe the behaviour of the different variants. In traditional (product-based) fingerprinting, there is usually a set of candidate models from which we have to compute an input sequence. By using a family-based signature of a system, we can potentially improve the fingerprinting process since we consider a more efficient signature. As with a set of product FSM signatures, we do not have knowledge of which states and transitions are equal while with an family-based signature the behavioural commonalities and differences between the products are annotated.

In this chapter, we propose methods for passive and active fingerprinting using the family-based signature. Currently, family systems often do not have this signature available. In Section 2.2.1.1, we discuss a technique to create the family-based signature from a set of FSM models. In Section 4.1, we use this technique to create a family-based signature which is used for fingerprinting systems. Next, in Section 4.2, we present a passive fingerprinting method using this family-based signature. With a passive fingerprinter, we observe a set of input/output traces of a running black-box system. As discussed in Section 2.3, This has the benefit that the system is operable and that the system is unaware that it is being fingerprinted. However, sometimes not enough traces are observed, which makes fingerprinting a single version impossible. To always fingerprint a system, we present multiple active fingerprinting approaches in Section 4.3. With these methods, we calculate a sequence based on the family-based signature. This sequence is executed on the black-box system and based on the output we determine the version. As we interact with the black-box system, the drawback is that the system knows it is being fingerprinted.

The family model creator, passive fingerprinter and active fingerprinters are implemented in Python and available on GitHub[1]. Appendix A provides usage instructions for the GitHub.

[1] `https://github.com/GianniM123/FFSM_fingerprint`

4.1 Family model creation

For a family model, we use the Feature Finite State Machine representation (Section 2.2.1). Since most families of systems do not have this model, we need a technique to create an FFSM. For this, we use the FFSM$_{diff}$ algorithm from Section 2.2.1.1. With this technique, we create an FFSM model from the different available variants of FSM representations.

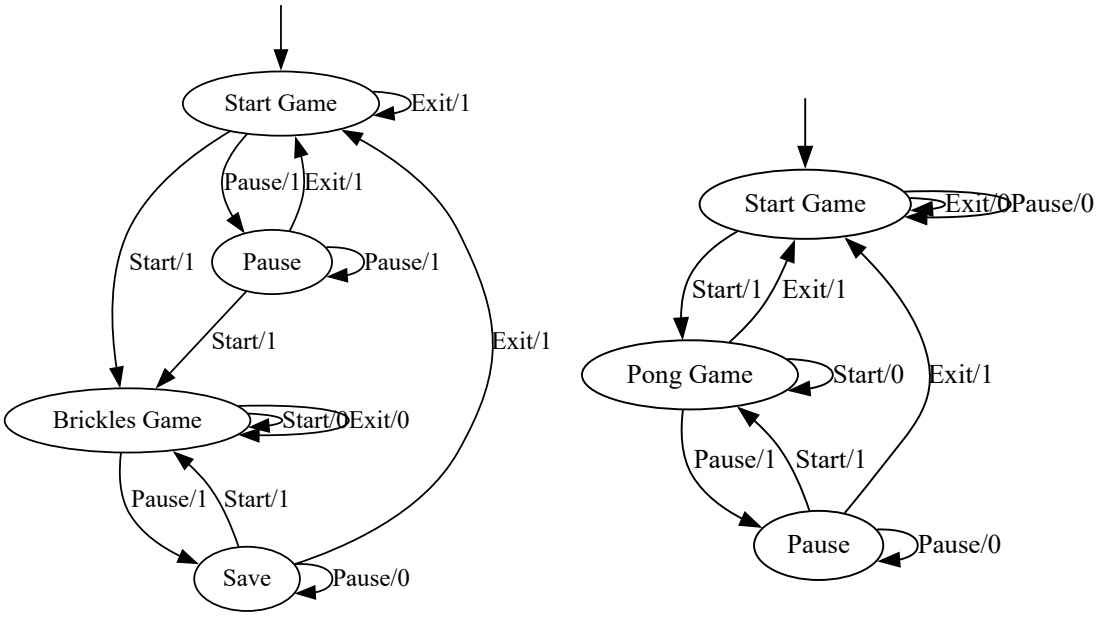
Another problem is that often there is no feature model available. Therefore, we need to create a feature model that can distinguish the variants in the FFSM model. By giving the variant a name or using the version number, we build a feature model where a single feature which is used to represent an entire system variant. To still be able to derive only correct system variants, a feature model is created where the different variants have an xor relationship. In this way, only one variant can be derived from the FFSM model.

In Figure 4.1, we see a feature model, expressed as a propositional formula, with an xor-relationship between the three variants B, P, and W and Figure 4.2 shows their representations. Figure 4.3 shows the corresponding FFSM representation created with the FFSM$_{diff}$ algorithm with the variants B, P and W.

When fingerprinting the variant of a black-box system, we need to analyse if a transition can be executed considering the current set of possible variants. By having a single feature as the feature representation for a variant, we enable a lightweight process for variability analysis. Normally, when the transitions and states are annotated with a feature representation, we need SAT-solvers to verify if a specific configuration can execute the transition. But now with the single feature representation, instead of using a SAT-solver, we can use set theory to analyse the variability. In set theory, an *and*, *or*, and *xor* can all be simulated, if we take the *intersection* of two sets, we simply simulate an *and* operation, and if we take the *union* of two sets, we simulate an *or* operation. For the *xor*, we simply take the symmetric difference between two sets. In practice, this would look like Example 4.1.

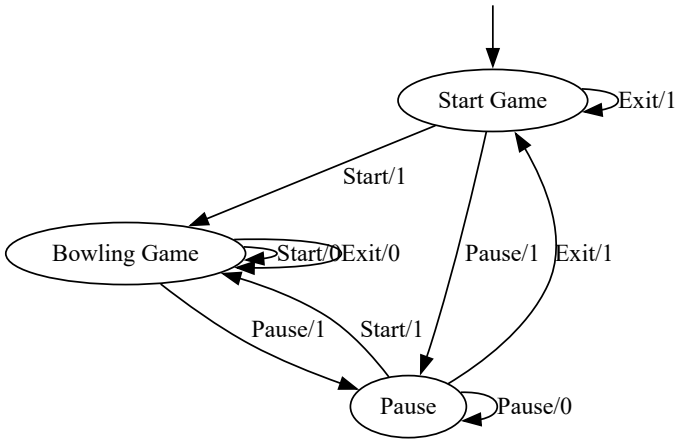
$$(B \wedge \neg P \wedge \neg W) \vee (\neg B \wedge P \wedge \neg W) \vee (\neg B \wedge \neg P \wedge W)$$

Figure 4.1: Feature model for game machine



(a) FSM for Brickles (B)

(b) FSM for Pong (P)



(c) FSM for Bowling (W)

Figure 4.2

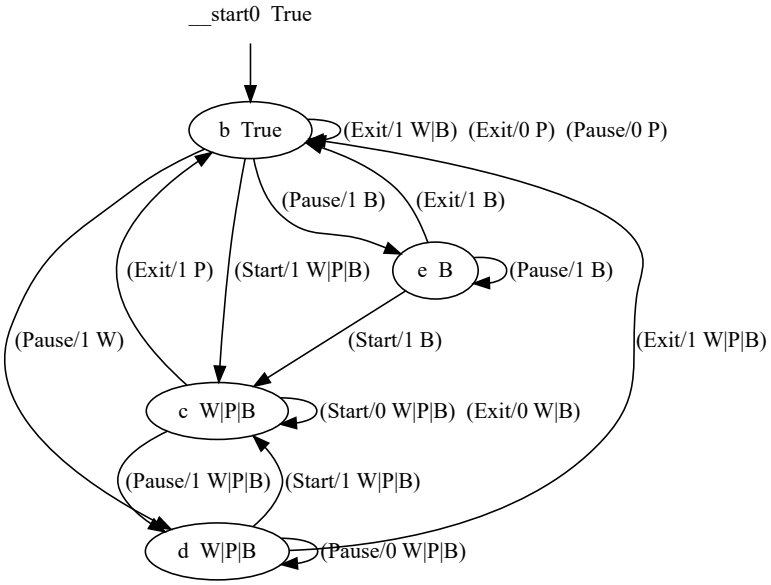


Figure 4.3: FFSM for a game machine

Example 4.1 *Consider the FFSM in Figure 4.3 and the configurations set* $\Lambda = \{W, P, B\}$.

If we then start in the initial state b *and want to execute the trace* $t = (Exit/1, Pause/1, Pause/1)$ *we analyse if this is possible with the following statement:*

$$\{W, P, B\} \cap \{W, B\} \cap ((\{W\} \cap \emptyset) \cup (\{B\} \cap \{B\}))$$

From this analysis, we conclude that it is possible to execute the trace if we use configuration B.

4.2 Passive family-based fingerprinting

When we are passively fingerprinting a SPL variant of a black-box system, we first observe an input/output trace from the system. This trace is then fed to the algorithm in combination with the family-based representation where different configurations exists in the set Λ and the transitions and states are annotated with the features. By traversing the corresponding transitions according to the input/output trace from the black-box system we cross out certain configuration options by the conjunction of the presence conditions. After this, we have a potentially smaller configuration subset and if this subset is a singleton, then we have successfully fingerprinted the system.

For passive fingerprinting there already exists an algorithm which is discussed in Chapter 2. By adapting the proposed algorithm to use an FFSM (Algorithm 2.2), we create the technique to detect the configuration of the black-box system. To do this, we adapt the algorithm on the following points:

- PEFSMs to FFSM, in Algorithm 2.2 they use a set of PEFSMs (line 1). We use the FFSM to passively fingerprint the models (line 1).
- We still add all the states from the model as an *uncertainty* but make a tuple of it with the conditional states that include the states and the respective feature selection of the conditional state (lines 5 to 7). Since we assume that one feature represents one configuration, we just add B, the set of configurations in which the state exists, to the tuple without first unifying it with the configurations Λ.
- When we loop over all the transitions in the model, we add a check with the current configurations if it is possible to take this transition (line 13). This is done by taking a set intersection of the current possible configurations and the presence condition of the transition.
- If it is possible to traverse a transition, we take the set intersection and add this in the *uncertainty* pair (line 14).
- As a last step in the algorithm we take the *union* of the different available configurations to return all the possible candidate configurations (lines 22 to 24).

When we apply the proposed changes to the algorithm, we end up with the algorithm of Algorithm 4.1. In Example 4.2 a running example of the algorithm is given.

Algorithm 4.1 Passive Group Matching - FFSM

Input: FFSM F
trace $T = \langle\langle I_0, O_0\rangle, \langle I_1, O_1\rangle, ..., \langle I_{L-1}, O_{L-1}\rangle\rangle$
Output: Configurations
uncertainty = {}
for all (s, B) **in** *F.states* **do**
    uncertainty.add((s, B)) ▷ We can start in any arbitrary state
end for
id = 0;
while *uncertainty* != {} **and** $id < L$ **do**
    new_uncertainty = {}
    for all $t\langle\langle S_{src}, B_{src}\rangle, \langle S_{dst}, B_{dst}\rangle, \langle I_t, B_t\rangle, O_t\rangle$ **in** *F.transitions* **do**
        for all (s, f) **in** *uncertainty* **do**
            if $(S_{src} == s$ **and** $I_{id} == I_t$ **and** $O_{id} == O_t$ **and** $can\_unify(B_t, f))$ **then**
                new_uncertainty.add($(S_{dst}$, $unify(B_t, f)))$
            end if
        end for
    end for
    uncertainty = *new_uncertainty*
    id = *id* +1
end while
configs = {}
for all (s, B) **in** *uncertainty* **do**
    configs = *configs*.union(B)
end for
return *configs*

Example 4.2 *(Running Algorithm 4.1) Let's consider the system of Figure 4.3 as our FFSM and consider the trace* $\langle\langle Start, 1\rangle, \langle Exit, 1\rangle, \langle Pause, 0\rangle\rangle$. *We first then initialize our* ***uncertainty*** *to* $\{(b, W|B|P), (c, W|B|P), (d, W|B|P), (e, B)\}$ *and then we are ready to start. We pick the first input/output combination of the trace, which is* $\langle Start, 1\rangle$, *and search for corresponding transitions from the* ***uncertainty***. *We repeat this until we have an empty* ***uncertainty*** *or have traversed all items from the trace. In Table 4.1 we see how the* ***uncertainty*** *develops itself after every input/output combination. As we see in the last row we only have configuration* P *and therefore we identified* P *as the black-box system that produced the trace.*

Inputs/Outputs	**Uncertainty**			
	$(b, W\|B\|P)$	$(c, W\|B\|P)$	$(d, W\|B\|P)$	(e, B)
Start/1	$(c, W\|B\|P)$	*No Start/1 transition*	$(c, W\|B\|P)$	(c, B)
Exit/1	(b, P)	-	(b, P)	$\{P\} \cap \{B\} = \emptyset$
Pause/0	(b, P)	-	(b, P)	-

Table 4.1: Passive family-based fingerprinting FFSM example

4.3 Active family-based fingerprinting

Currently, there is no family-based approach known for active fingerprinting deterministic black-box systems. While it is possible to derive and then analyse each valid product from the FFSM, this approach does not benefit of the family-based signature. Therefore we introduce three new ways of fingerprinting, where we explore the Online Machine Separation method, as discussed in Section 2.3, and state distinguishing sequences, as shown in Section 2.1.1.

First, in Section 4.3.2, we propose a family-based Online Machine Separation [24] which separates two configurations of Λ at one time instead of two PEFSMs. This method gives us the benefit of using the family-based signature but does not lead to the efficiency increase in terms of the sequences, as we now need to calculate $\mathcal{O}(\Lambda^2)$ sequences. But by using this method as our baseline, we can compare our potentially better fingerprinters to this solution to display the decrease in the number of inputs and resets. For proposing better fingerprinters, we want to have fewer sequences to separate all the different configurations Λ than $\mathcal{O}(\Lambda^2)$ sequences. As discussed in Section 2.1.1, state distinguishing sequences can find the initial state of a Mealy machine by splitting all states based on a single sequence. Here a regular sequence which separates all the states based on the output and finds the initial state is called a *preset* distinguishing sequence (PDS). Since we want to split all configurations in our family-based signature, we need to adapt the PDS into the *configuration-based* PDS (CPDS), this is described in Section 4.3.3. A CPDS is thus a regular sequence which separates all the configurations Λ. In the theory of state identification, they also consider a more complex sequence, the *adaptive* distinguishing sequence (ADS). This method is not a real sequence but a tree where the input depends on the seen output. In state identification, the ADS is considered more powerful as it can exist more often and is often shorter than the PDS. Since the ADS can calculate shorter sequences, we also want to propose this as a fingerprinting method. Therefore, in Section 4.3.4, we propose an algorithm to calculate a *configuration-based* ADS where we separate all the different configurations.

To introduce these algorithms, we require three assumptions on the family-based signature and black-box system. Next, we need a transfer and output function for the FFSM just as we have for the FSM. At last, we discuss the *configuration-based* distinguishing sequence, this is the separating sequence decorated in a tree to have a uniform representation for all the different active fingerprinters. All these topics are discussed in Section 4.3.1.

4.3.1 Preliminary

Before we introduce our family-based fingerprinters, we first discuss the three assumptions we made on the family-based signature and black-box system. These are common assumptions for running model-based testing techniques, as in Kirchen [19] which we adapted to distinguish configurations.

1. We assume that all configurations Λ are non-equivalent, i.e. each pair of configurations will produce a different output for an existing input sequence

2. We assume that at the beginning of the search for a sequence, the black-box system is resettable to its initial state.

3. We need to make sure that the FFSM is *complete* (Definition 2.7). In short, an FFSM is *complete* when for every conditional state and all configurations all the different input symbols have a transition.

When an FFSM is not complete, it can be the case that there exists a trace for configuration ξ while it does not exist for configuration ξ'. This can have influence on the separation of two configurations. For example, it can be the case that configuration ξ has an input alphabet i_ξ that is larger than the input alphabet $i_{\xi'}$ of configuration ξ', i.e. $i_{\xi'} \subset i_\xi$. When now configuration ξ implements the same behaviour as ξ' using the input alphabet $i_{\xi'}$, it is impossible to separate ξ and ξ' from each other since for every trace made from $i_{\xi'}$ ξ and ξ' return the same output. However, when we use the alphabet for which ξ' does not have an implementation, i.e. $i_\xi \setminus i_{\xi'}$, it becomes possible to separate them as ξ' will not return an output. To solve this issue, we

make the FFSM *complete* when this is not the case. For this, there are different approaches, a *demonic* or an *angelic* approach [9]. With a *demonic* approach, an undefined input is handled as a catastrophic failure. This means that the system moves to an error state which it can not recover from. While with the *angelic* approach, the undefined inputs are ignored. This means that the system essentially stays in the same state. Since we assume that the black-box system will ignore the inputs and stay in the same state, we use the *angelic* approach. This is done by creating a self-loop in the conditional state with the input and an ϵ as output. From our example, it would mean that we update the transitions with alphabet $\alpha = i_\xi \setminus i_{\xi'}$ for ξ' by adding a self-loop with input from α and output ϵ for every conditional state where ξ' is seen.

Now, to be able to create the approaches, we need an output and transfer function for the FFSM. The output function is of the form $\lambda : (S, \xi) \times I \to \{(O, \xi)\}$ and the transfer function is of the type $\delta : (S, \xi) \times I \to \{(S, \xi)\}$. Here S is the current state, ξ is a set of current configuration, I is the input, and O is the output. By default, these functions can also handle a block B of (S, ξ). This is handled in the following manner:

$$\lambda(B, x) = \{\lambda(s, x) | s \in B\}$$

Both functions use a set of configurations ξ and a state S and output a new set of configurations. For the input, we use a set of configurations ξ and state S instead of a conditional state C, this enables us to execute the state with fewer configurations as specified in the conditional state C and is needed when we earlier refined the configurations with an input. For the output, we always obtain a set of new configurations. In this way, we obtain information on which configuration produced which output or moved to which state. Below, examples of the functions are given for Figure 4.3.

$$\lambda((b, \{W, P, B\}), Pause) = \{(1, \{W, B\}), (0, \{P\})\}$$

$$\delta((b, \{W, P, B\}), Pause) = \{(d, \{W\}), (e, \{B\}), (b, \{P\})\}$$

For every approach which we propose, the result is a *configuration-based* distinguishing sequence (CDS). This is a tree labelled with the inputs, outputs, and corresponding configuration. At the root of the tree, we have not distinguished any of the configurations and assume that the system is in its initial state. This means that when we want to execute the tree, the black-box system needs to be reset to its initial state. Now we execute the label of the root on the black-box system and observe the output. By following the edge with the same output, we end up in a new node. We then repeat the process until we reach a leaf node indicating the configuration of the black-box system. In Definition 4.1, the formal definition of the *configuration-based* distinguishing sequence is given, and in Example 4.3, a running example of a CDS can be found.

Definition 4.1 *A configuration-based distinguishing sequence (CDS) of a Feature Finite State Machine $M = (F, \Lambda, C, c_0, Y, O, \Gamma)$ is a rooted tree T such that:*

1. *The number of leaves of T equals the cardinally of Λ, thus every leaf has one single configuration $\xi \in \Lambda$.*
2. *The internal nodes of T are labeled with the input symbols from I.*
3. *The edges of T are labeled with the output symbols from O.*
4. *Edges emanating from a common node are labeled with distinct output symbols.*
5. *For each leaf $u \in T$, if ξ_u is the label of u, x_u and y_u are respectively the input and output sequences formed by the concatenation of the node and edge labels on the path from the root to the leaf u, then $\lambda((s_0, \xi_u), x_u) = y_u$.*
6. *The length of the CDS is the depth of its corresponding tree T.*

Note:

- *All configurations Λ need to be non-equivalent.*

Example 4.3 *(Running example for Figure 4.4) Let's consider an FFSM with the configurations $\Lambda = \{P, G, W, B\}$. From this FFSM we compute the CDS of Figure 4.4.*

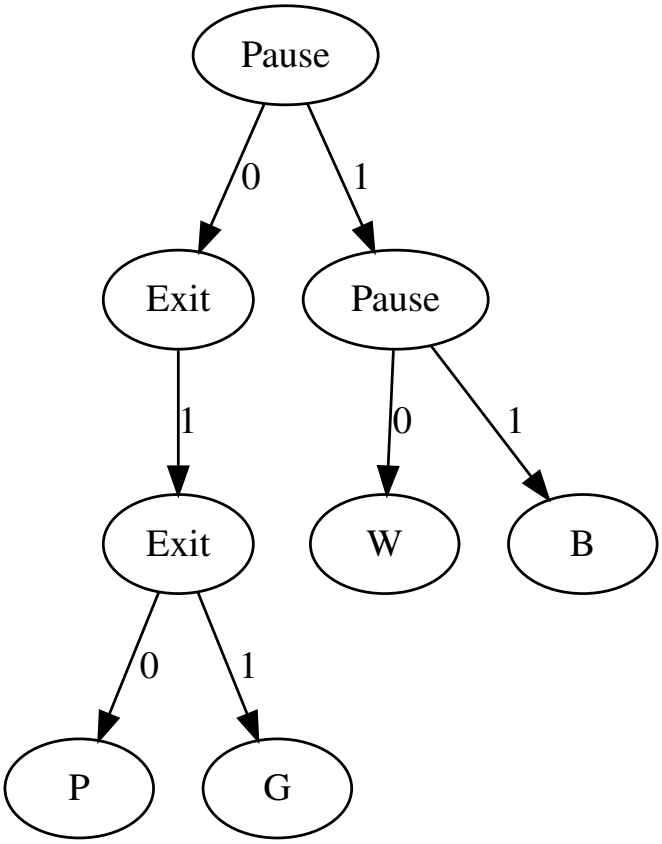


Figure 4.4: configuration-based distinguishing sequence example with $\Lambda = \{P, G, W, B\}$

Now we want to use this CDS to fingerprint a black-box system. At first, we need to guarantee that the black-box system is in its initial state. When this is guaranteed, we apply the input of the root of the tree on the black-box system. In this case of Figure 4.4 this is Pause. After we have applied this input, we observe the output the system has returned, which for the current black-box system would be a 0*. Now, we follow the edge from the root node which corresponds to* 0 *and end up in a new node which has as input Exit. Then, we apply Exit and observe output* 1 *of the system and by following the correct edge we now again need to apply Exit. From the system, we now observe* 1 *and this effectively means that we have ended up in the leaf G, indicating the configuration set in our black-box system.*

4.3.2 Family-based Online Machine Separation

As discussed in Section 2.3, Shu and Lee [24] proposed a method to split a set of candidate group fingerprints C by calculating separating sequences for a pair of models. By switching from a set of candidate group models to a family-based representation for fingerprinting the system, we still use this algorithm as a basis. Instead of now splitting the different candidate models C, we split the different configurations Λ from the FFSM. In Algorithm 4.2 we see the updated algorithm. Since the algorithm only obtains a fingerprinting set F_s, we need to do a post-processing step to create a configuration-based distinguishing sequence. This is done by executing the sequences on the FFSM and resetting the system whenever we start with a new sequence. Now, we label the resets, inputs, and outputs on the edges and end up with the result in Figure 4.5.

For calculating the separating sequences, we know that the systems are in their initial state at the beginning. This makes the search for a separating sequence more less involved than if the initial state was not known. For finding this sequence, we do a Breadth-first search by applying every input on the initial states and observing the outputs and newly reached states. If there exists a given input for which we have a different output we stop and return the input. Otherwise, we apply the inputs on the newly reach states and observe the new outputs and new reach states. We then combine the earlier done inputs with the newly done inputs to create a sequence. We eventually stop when we either found a sequence that does not give the same output or we stop if all combinations of states have already been explored. In the last case, a separating sequence does not exist. After we have applied such a sequence on the black-box system, we are not in the initial state anymore. By applying the input $RESET - SYS$ the black-box system will return to its initial state.

Another change in the algorithm is that we do not check if a sequence exists for a given configuration. With the original Algorithm, the sequence can be defined for signature M_i and signature M_j, but not for another signature M_k in the set. When this happens, M_k is added to all the newly created partitions. As we made our family-based signature *complete*, all the inputs have a transition for all the configurations. This means that all sequences are defined for all configurations and we do not need to check if a sequence exists.

Algorithm 4.2 Active Group Matching - Family model

```
1:  Input: FFSM F
2:  Output: fingerprint set F_s
3:  F_s = {}
4:  partition = {Λ}
5:  while partition.size < Λ.size do
6:      select C_i and C_j in same partition set
7:      calculate separating sequence SEQ for C_i and C_j
8:      for all set S in partition do
9:          initialize map as an empty map from {F.outputs}* to POW(S)
10:         Remove S from partition
11:         for all x in S do
12:             map(f_output(C_x, SEQ))∪ = {x}
13:         end for
14:         for all s* in range(map) do
15:             if s* ∉ partition then
16:                 partition = partition ∪s*
17:             end if
18:         end for
19:     end for
20:     F_s = F_s ∪ {SEQ}
21: end while
22: return F_s
```

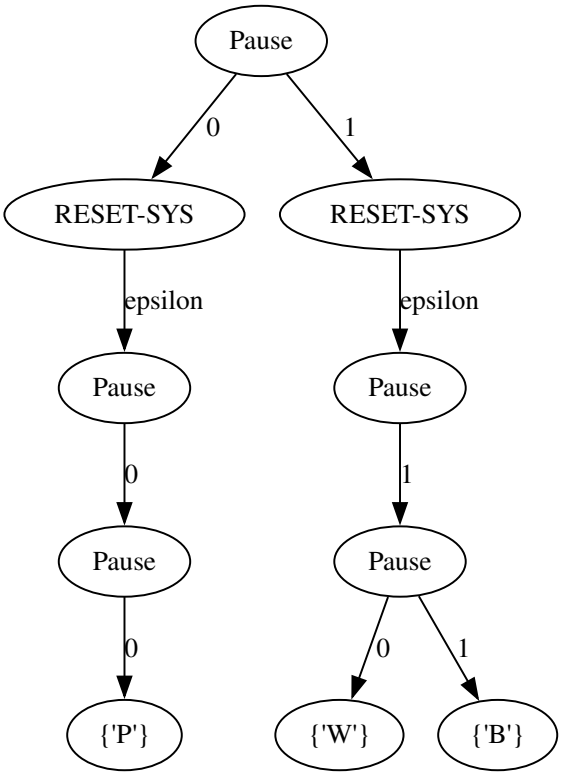


Figure 4.5: Online Machine Separation result example for Figure 4.3

4.3.3 Configuration-based Preset Distinguishing Sequences (CPDS)

In the previous section, we discuss a family-based Online Machine Separation approach. With this approach, the configurations Λ are pairwise separated from each other, which requires calculating $\mathcal{O}(\lambda^{\in})$ sequences. As discussed in Section 2.1.1.1, a Preset Distinguishing Sequence (PDS) is a single sequence which for a set of states produces a different output. By adapting this theory, we can separate all Λ with fewer sequences than the family-based Online Machine Separation.

At first, we lift the notion of the uncertainty to the scope of FFSMs so it is possible to generate a CPDS. This is given in Definition 4.2.

Definition 4.2 *For a given FFSM $M = (F, \Lambda, C, c_0, Y, O, \Gamma)$ and x an input sequence, the product and state uncertainties of M with respect to x are defined as follows:*

1. ***Product uncertainty*** *$\pi(x)$ is the partition $\{K_1, K_2, \ldots, K_r\}$ of Λ created by applying $\lambda((s_o, \Lambda), x)$ and combining the configurations that have the same output.*
2. ***State uncertainty*** *The state uncertainty can be calculated in the following manner:*

$$\sigma(x) = \{\delta((s_0, \theta), x) | \theta \in \pi(x)\}$$

This *product uncertainty* returns the current split in configurations after applying inputs. For example, when we consider the model from Figure 4.3 and apply *Exit* we get the following:

$$\pi(Exit) = \{\{W, B\}_1, \{P\}_0\}$$

This means that if on input *Exit* we get the output 0, we know that the configuration we are talking to consists of P. While if the output is 1, we know that it is possibly W or B. Note that the outputs are not actually provided by $\pi(x)$ but are added to make it more clear. From this, we see that if we have an input sequence x and we reach the discrete partition we know that x is a

CPDS for the FFSM. Here the discrete partition means that we encounter blocks that all consists of one configuration.

With the *state uncertainty*, we keep track of which state we have reached after applying the input and which configurations hold for the reached state.

$$\sigma(Pause) = \{\{(d, W), (e, B)\}_1, \{(b, P)\}_0\}$$

This tells us that after a sequence of *pause*, we distinguish P but under the same output W and B reach different states. This information can be used to define the equality of a partition or to pick an input that distinguishes states d and e and therefore W and B.

A classical approach to finding the PDS is done by calculating a successor tree [18]. This is a tree in which the nodes are labelled with the split states and the edges are all the inputs possible in the model. Another method to find the PDS is the super graph method [19], the main difference between these two is that the super graph is a finite structure and only considers valid inputs. From these two methods for finding a PDS, we define our **configuration successor tree**. An example of this tree can be found in Figure 4.6.

Definition 4.3 *a **configuration successor tree** for a given Feature Finite State Machine $M = (F, \Lambda, C, c_0, Y, O, \Gamma)$ is a tree T were the following properties hold:*

1. *Each node of T is labeled by the **product uncertainty**.*
2. *Each node of T is holds the **state uncertainty** and the corresponding output to the input sequence which is spelt by the path to the node.*
3. *Each node of T considers all the inputs for the current configurations.*
4. *Each node of T is only added in the tree if there does not exist another node in T with the same **state uncertainty**, i.e. $\forall n, n' \in T : \sigma(n) = \sigma(n') \leftrightarrow n = n'$.*
5. *The tree ends when either (a) a sequence is found when we reach the discrete partition of Λ or (b) no new combinations of **state uncertainty** can been explored.*

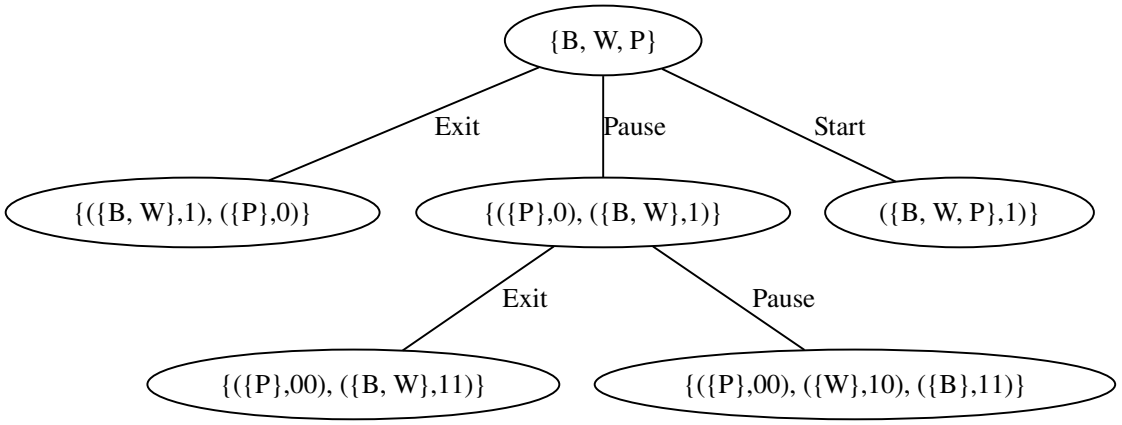


Figure 4.6: Configuration Successor Tree

By gradually building the configuration successor tree we can find a CPDS. As we see in Figure 4.7 we see that after applying the sequence $\langle\langle Pause\rangle, \langle Pause\rangle\rangle$ the 3 different configurations are distinguished. To find the shortest PDS, one can run a breadth-first search on the super-graph. Therefore we have designed an algorithm that follows the breadth-first search principle. It can be found in Algorithm 4.3. The algorithm starts at the root of the tree, this is represented by σ_0 on line 3. We here define that we have one large split of Λ, that we are in the initial state with all Λ

and currently have not applied any inputs (ϵ). To keep track of which σ we have already seen and which σ we still need to explore. We initialize a set E and E' (lines 4 and 5). With lines 7 and 8 we then decide if have explored new nodes during an iteration. If this is the case we continue and otherwise we stop. If we continue with the algorithm, we pick all σ we did not explore yet and apply all the inputs (lines 9 to 12). The last step is then to check if we have found the discrete partition, in which case the algorithm stops (lines 13 and 14), or we check if we have already seen the new σ' before. If we have not seen it we add it in E so we explore it in the next iteration (lines 15 to 17).

Note that at the end of this algorithm, we have obtained the input sequence and not yet the CPDS. But by applying the input sequence on the FFSM and observing their corresponding output symbols we build the CPDS. The final result for Figure 4.6 is given in Figure 4.7.

Algorithm 4.3 Find CPDS

```
1:  Input: FFSM F
2:  Output: CPDS
3:  σ_0 = ({Λ}, (s_0,Λ), ε)    ▷ Λ is the set of product configurations, s_0 the initial state and ε the
    empty sequence
4:  E = {σ_0}
5:  E' = {}
6:  Mark σ_0 as non-treated
7:  while E != E' do
8:      E' = E
9:      for all non-treated σ ∈ E do
10:         for all input a in F.alphabet do
11:             sequence = σ · seq + a
12:             σ' = (π(sequence), σ(sequence), sequence)
13:             if σ' · product is discrete partition then
14:                 return sequence as CPDS
15:             else if σ' · state not in E then
16:                 add σ' in E
17:                 mark σ' as non-treated
18:             end if
19:         end for
20:         mark σ as treated
21:     end for
22: end while
23: return "No CPDS exists"
```

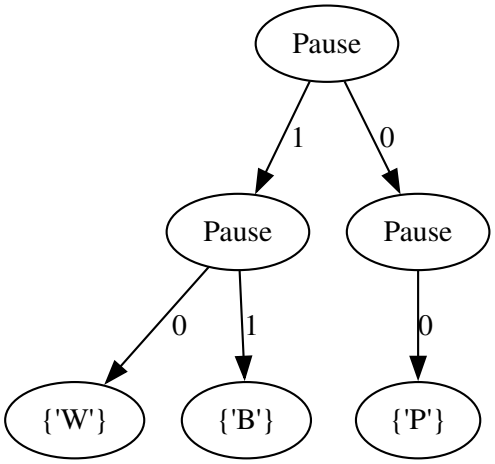


Figure 4.7: CPDS result example for Figure 4.3

4.3.4 Configuration-based Adaptive Distinguishing Sequences (CADS)

With a newly introduced method to find Configuration-based Preset Distinguishing Sequences, we would like to see the possibility of a Configuration-based Adaptive Distinguishing Sequences. This is because the regular ADS is often shorter than the regular PDS. As this also holds for the Configuration-based approaches, we can again improve the fingerprinting sequence.

To derive a CADS from the FFSM we first introduce an intermediary structure, the configuration splitting tree. In Definition 4.4 the formal definition is given and an example can be found in Figure 4.8.

Definition 4.4 *A **configuration splitting tree** associated with a Feature Finite State Machine $M = (F, \Lambda, C, c_0, Y, O, \Gamma)$ is a rooted tree T where:*

1. *Each node of T is labeled by a subset of Λ and the root T is labeled with Λ*
2. *The label of a node T is the union of its children's labels.*
3. *The internal of a node T is labeled with a set of states s and a set of configurations c such that $s \subseteq C$ and $c \subseteq \Lambda$.*
4. *Each edge of T is labeled with an input symbol from I and output symbol from O.*
5. *Each edge from a node must have the same input symbol from I and a different output symbol from O*

*Let $\omega(T)$ be the set of configurations formed by the leaves of the tree T. T is a **complete configuration splitting tree** if $\omega(T)$ is the discrete partition of Λ.*

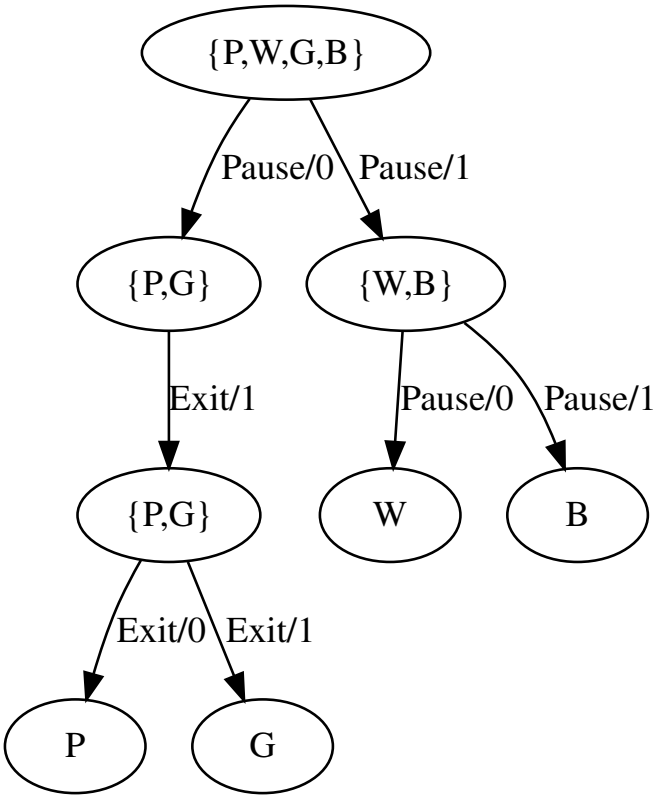


Figure 4.8: Complete configuration splitting tree

To now derive such a tree, we need to design a new algorithm as the algorithm which is proposed to derive the "normal" splitting tree for splitting states is not compatible with the configuration splitting tree. Since we do not already have an algorithm for the splitting tree, we could also draft an algorithm that immediately finds a configuration-based distinguishing sequence. But in fact, this would be even harder due to the loss of the current split of configurations. In this section, we present an algorithm which consist of three steps: explore splitting trees, combine trees and derive splitting tree.

4.3.4.1 Explore splitting trees

To explore splitting trees, we follow a breath-first-search approach. We start in the initial state considering that all configurations are not distinguished and start applying the inputs. For every applied input, we observe the different possible outputs and decorate this in a configuration splitting tree which belongs to the explored node.

4.3.4.2 Combine trees

The goal of the algorithm is to find a complete splitting tree, therefore we combine the single trees in a graph. But instead of combining all the splitting trees, we are only interested in the trees that contain useful information. As useful information, we are considering two types:

1. *A splitting tree has a single configuration leaf.*

   According to the rules of a complete splitting tree, all the leaves need to be a single configuration. Therefore all trees that have a single configuration are interesting and are combined into the graph.

2. *An input for which the configurations split, but all the new nodes are already in the discover queue or were previously discovered.*

For this rule, we need to understand that for a given input a in conditional state c a set of configurations ξ can split to ξ_1 and ξ_2 where $\xi = \xi_1 \cup \xi_2$ holds. With input a, ξ_1 is obtained in conditional state d while ξ_2 is obtained in conditional state e. Now, we reach (ξ_1, d) and (ξ_2, e) from (ξ, c), but there can also exists other paths to (ξ_1, d) or (ξ_2, e). It can be the case that we reach (ξ_1, d) from conditional state f and (ξ_2, e) from conditional state g. When we consider the case that we first explored the conditional states g and f, we already have obtained the (ξ_1, d) and (ξ_2, e). If we now explore conditional state c, we again obtain (ξ_1, d) and (ξ_2, e). Since we discovered them already, we do not need to add them to the queue to explore them further. But in this case, we would lose information about how to reach (ξ_1, d) and (ξ_2, e) as there are two potential paths. By adding the splitting tree with such a split to the combined splitting graph, we potentially add a path to a single configuration. Since not every split of configurations adds a path to a single configuration, but as discussed in Section 4.3.4.3, the information is removed when it is not needed.

4.3.4.3 Derive splitting tree

Now that we have an procedure to create the combined graph, we need to have a method to check if we can derive a complete splitting tree out of the graph. For this, we have defined the following steps:

1. Every leaf needs to be a singleton configuration. If this is not the case we remove the leaf. (Rule 1 of Definition 4.1)
2. For every input of a node, the children labels are the union of his parent. If this property does not hold we remove the input edges of the node. (Rule 1 and 2 of Definition 4.4)
3. If multiple inputs exists at a node, pick one input and remove the other inputs. (Create a tree)
4. Remove all nodes, except for the root, which do not have an incoming edge. (Clean up)

At the end of this process, we either have obtained a complete configuration splitting tree or an empty tree. If we have found the complete tree we stop the algorithm and otherwise we continue the search.

4.3.4.4 Algorithm

In Algorithm 4.4, we present the pseudocode for finding a complete splitting tree which has as its input a deterministic FFSM $M = (F, \Lambda, C, c_0, Y, O, \Gamma)$. At the start of the algorithm, we define the initial σ_0 (line 3) which consists of the feature selection Λ, the current state, and the graph with a root node and is added in the *explore* queue from which we pick the next σ to explore. Next to this, we define a *seen* list in which all the previously explored σ's are stored and at last, we define the *combined_graph* for combining the splitting trees. After the initialization, we start looping over the elements from the *explore* queue and add them to the *seen* list (lines 7 to 9). In the next step, we loop over the input alphabet, copy the available tree, and create a set for the new σ's which we find and calculate the outputs (lines 10 to 13). By copying the tree, we make sure that for every input a new tree is created and thus the properties of the configuration splitting tree hold. In lines 14 to 22, we iterate over the different possible outputs and their corresponding new configurations. With this information, we decide the new state to which we have travelled and only decorate this information inside the tree if the new state is not already in the tree. In the end, we create a new σ with the new configurations, state, and tree. In line 24, we check if we have obtained a split in configurations. From lines 25 until 34, we first update the σ with the complete splitting tree and check if we have found a single configuration. If this is the case, we combine the new tree with the existing graph and run the *obtain_splitting_tree* function to obtain the complete configuration splitting tree. After this operation, we have either obtained the complete splitting tree, which means we return the tree, or we have found an empty tree which means we continue with the algorithm. If we have not found a single configuration, in lines 36 to 40, we check if we

have already seen the state configuration. Otherwise, we add it to the *explore* queue. Now the last step of the algorithm is to check if *interesting_tree* is true (lines 42 to 44). When this is the case, we have obtained multiple nodes but they are not explored further. Therefore we combine the new tree with the combined graph.

After this algorithm, we strip the input and output from the edges. The input is then placed at the source node of the edge and the output is placed on the edge. By repeating this for every node which is not a leaf, we end up with the CADS. An example of this whole algorithm can be found in Example 4.4.

Example 4.4 *(Building combined_graph)*
In Figure 4.9 we show the growth of the combined graph for the FFSM of Figure 4.3.

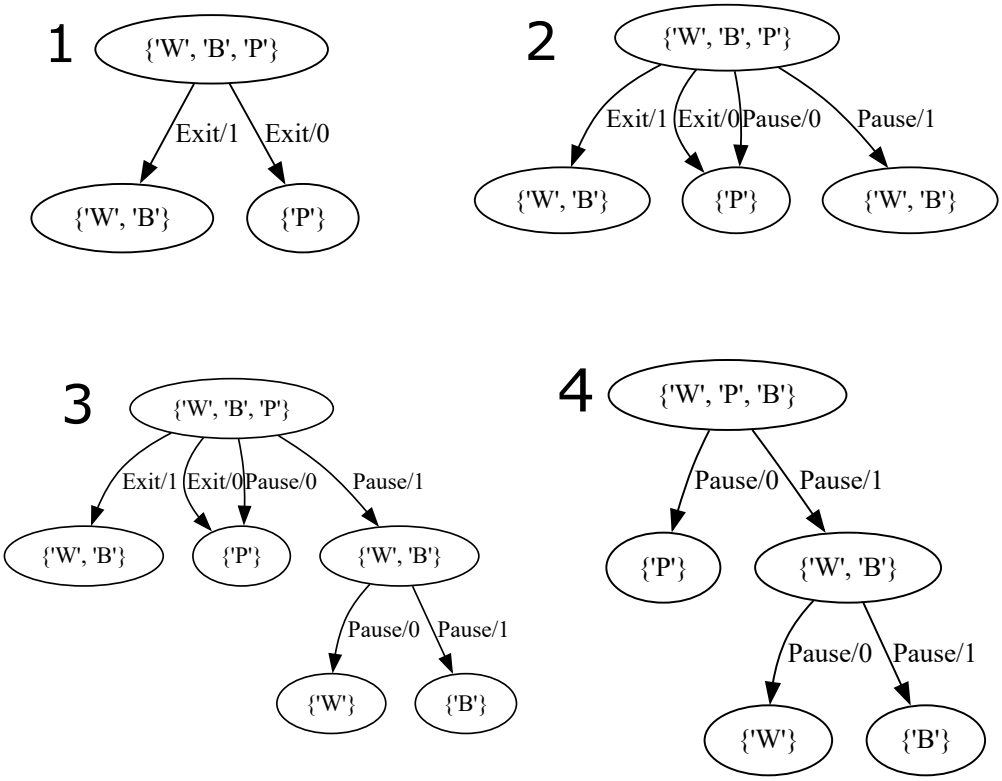


Figure 4.9: Building the graph (*combined_graph*)

In the first step, Exit is applied to the system. This reveals the single configuration P and to keep it conform to the standard of the splitting tree, the other split $\{W, B\}$ is also inserted. In the second step, we apply Pause and also observe the same single configuration P. In step three we explore the single configurations W and B by applying $Pause$. Since now the property $\Lambda \subseteq \omega(T)$ holds, we try to obtain the splitting tree by the earlier defined rules. In the last step (4), we have obtained the complete configurations splitting tree ($\Lambda = \omega(T)$).

Now that we obtained the complete configuration splitting tree, we lift the inputs to the nodes and end up with the CADS in Figure 4.10.

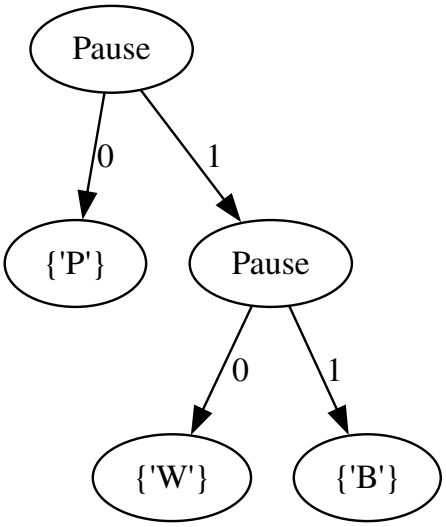


Figure 4.10: CADS result example for Figure 4.3

Algorithm 4.4 Build configuration splitting tree

Input: FFSM F
Output: CADS
$\sigma_0 = (\{\Lambda\},(s_0,\Lambda), \{(s_0,\Lambda)\})$ ▷ Λ the set of product configurations, (s_0,Λ) the current state and $\{(s_0,\Lambda)\}$ a graph with a root node which contains the configurations and the initial state.
explore $= \{\sigma_0\}$ ▷ The equality for σ is implemented on (σ·*config*, σ·*state*)
seen $= \{\}$ ▷ The seen σ
combined_graph $= \{\}$ ▷ The graph in which we combine the trees of the single configurations
while *explore* **is not** $\emptyset$ **do**
    σ = *explore*.pop
    seen = *seen* $\cup$ σ
    for all a **in** F.*alphabet* **do**
        new_tree = copy(σ· *tree*)
        new_sigmas = $\{\}$
        outputs = $\lambda(\sigma$·*state* $, a)$
        for all (o,ξ) **in** *outputs* **do**
            $s = \delta((\sigma$·*state*· *state* $,\xi), a)$
            if s **not in** *new_tree* **then**
                ▷ Only add the node and edge if the node is not already inside the tree
                new_tree.add_node(s)
                new_tree.add_edge(ξ, s, a/o)
            end if
            σ' = (ξ, s,*new_tree*)
            new_sigmas = *new_sigmas* $\cup$ σ'
        end for
        interesting_tree = *outputs.length* > 1
        for all σ' **in** *new_sigmas* **do**
            σ'·*tree* = *new_tree* ▷ Make sure all the σ have a valid splitting tree
            if σ'·*configs* **is** single configuration **then**
                interesting_tree = False
                seen = *seen* $\cup$ σ'
                combined_graph = *combined_graph* $\cup$ *new_tree*
                splitting_tree = obtain_splitting_tree(*combined_graph*)
                if ω(*splitting_tree*) **is the** discrete partition **then**
                    return *splitting_tree*
                end if
            else
                if σ' **not in** *seen* **and** σ' **not in** *explore* **then**
                    explore = *explore* $\cup$ σ'
                    interesting_tree = False
                end if
            end if
        end for
        if *interesting_tree* **is** True **then**
            combined_graph = *combined_graph* $\cup$ *new_tree*
        end if
    end for
end while
return "No CADS exists"

Chapter 5

Experimental comparison of fingerprinters

Chapter 4 discusses different approaches to fingerprint systems using a family-based representation. In this chapter, we compare the different methods against each other on the basis of the following evaluation questions (EQ's):

- **EQ1:** How does the family-based Online Machine Separation perform against the product-based version?
- **EQ2:** How much more efficient is the Configuration-based Adaptive Distinguishing Sequence over the Preset?
- **EQ3:** How does family-based Online Machine Separation perform against the Configuration-based Adaptive Distinguishing Sequence (CADS)?

Note that we do not explicitly compare CPDS to the family-based Online Machine Separation. This is because we expect the CADS to outperform the CPDS, a conjecture we study in EQ2.

To compare the different approaches against each other, we need a family system which includes multiple systems. For this, we used the set of OpenSSL models presented by Janssen[1] [15]. These models were originally not conceived from a SPL but where learned from the different published versions. As the current approaches can only separate models which have different behaviour, we need to make sure that the models are not testing equivalent [5]. To do so, we test the models using the AutomataLib[2] test equivalence function, which returns "true" if two versions can be separated from each other using a sequence or "false" in case there does not exist a sequence which can separate them. In Appendix A, we discuss where the program is available and how to use it. After running all the models with the equivalence checker, we found out that there are 16 different OpenSSL representations.

5.1 Online Machine Separation

In Section 4.3.2, we propose a family-based Online Machine Separation which is a adaptation of the product-based approach discussed in Section 2.3. Both techniques calculate a set of separating sequences and return them in the form of a Configuration-based distinguishing sequence. To see how the family-based approach performs, we compare it to the product-based approach. By comparing these two methods, we answer EQ1. Note that for the actual implementation, we adapted the product-based Online Machine Separation algorithm to separate FSM signatures instead of PEFSM signatures.

[1] https://github.com/tlsprint/models/tree/master/models
[2] https://learnlib.de/projects/automatalib/

5.1.1 Methodology

As discussed in the introduction, comparing the methods in terms of efficiency happens on two places: The time to generate the sequence and the efficiency of the sequence. Since the result of these algorithms is the same but only the implementation differs, we only need to compare the time to generate the sequences. Therefore we compare family-based and product-based implementation on the computational costs to generate these sequences in terms of execution time. We designed an experiment to see if the execution time correlates with the number of input models and the time it takes to generate the sequences by running the experiments from the minimum (2) to the 16 distinct available versions of the OpenSSL models. Instead of just picking x random distinct models and running the different fingerprinters with these models, we sort the model based on their version indicator in ascending order. For every option, we then execute the fingerprinters 100 times.

To compare the results we create a box plot that contains the CPU wall time of the calculation of the sequences. Next to this, we perform statistical hypothesis testing using the Mann-Whitney U to determine statistical significance. If we obtain statistical significance, we calculate the effect size using Vargha and Delaney's metric [3]. This means the effect size tells us the percentage of time one method is better than another. For example, when we compare method A to method B and the effect size is 0.7, we know method B has higher results in 70% of the cases. In terms of execution time, this means that the method B is slower than A in 70% of the cases.

5.1.2 Results execution time sequence generation

When we compare the family-based approach with the product-based approach, we see that the product-based is faster than the family-based. Especially when looking in Table 5.1, we see that the p-value is less than 0.05 for all the comparisons, confirming statistical significance and that the effect size most of the time is even equal to one, indicating a very large effect. This means that the family-based implementation has a higher execution time than the product-based implementation in most cases. When we look at the box plot in Figure 5.1, we see that indeed the execution times are lower for the product-based fingerprinter. This is because, for every input, we need to do a set intersection operation to verify if it is possible to take the transition, which causes more overhead. However, when we look at the absolute difference between the two fingerprinters, we see that on average the difference is no more than ten milliseconds. Also, since the highest execution time of the family-based fingerprinter is around 21 milliseconds[3], it still is an approach which has practical usage as and we have the benefit of a compact signature where behavioural similarities are explicitly denoted.

For convenience, we also executed the same experiments with the OpenSSL models sorted in descending order and listed them in Appendix B. For the descending order, we also see that the product-based implementation is faster, except for the case where we have three models, then the family-based fingerprinter is faster.

[3]Raw results: `https://github.com/GianniM123/FFSM_fingerprint/blob/main/benchmarks/benchmark_shulee_openssl.csv`

Nr of versions	p-value (Mann-Whitney U)	Effect size Product-based vs. Family-based	Magnitude	Avg. time (s) Product-based	Avg. time (s) Family-based
2	5.794e-07	0.7046	Medium	0.000947	0.001013
3	2.561e-34	1	Large	0.000685	0.001483
4	7.355e-33	0.9887	Large	0.001233	0.001677
5	2.562e-34	1	Large	0.001167	0.002568
6	2.562e-34	1	Large	0.001591	0.003605
7	2.562e-34	1	Large	0.001907	0.004176
8	2.562e-34	1	Large	0.002455	0.005640
9	2.562e-34	1	Large	0.002917	0.006550
10	2.562e-34	1	Large	0.003475	0.007924
11	2.562-34	1	Large	0.004324	0.008773
12	2.562e-34	1	Large	0.005194	0.011082
13	2.562e-34	1	Large	0.005124	0.012023
14	2.562e-34	1	Large	0.006003	0.012944
15	2.562e-34	1	Large	0.006290	0.013994
16	2.562e-34	1	Large	0.006701	0.014605

Table 5.1: Execution time comparison: product-based vs. family-based

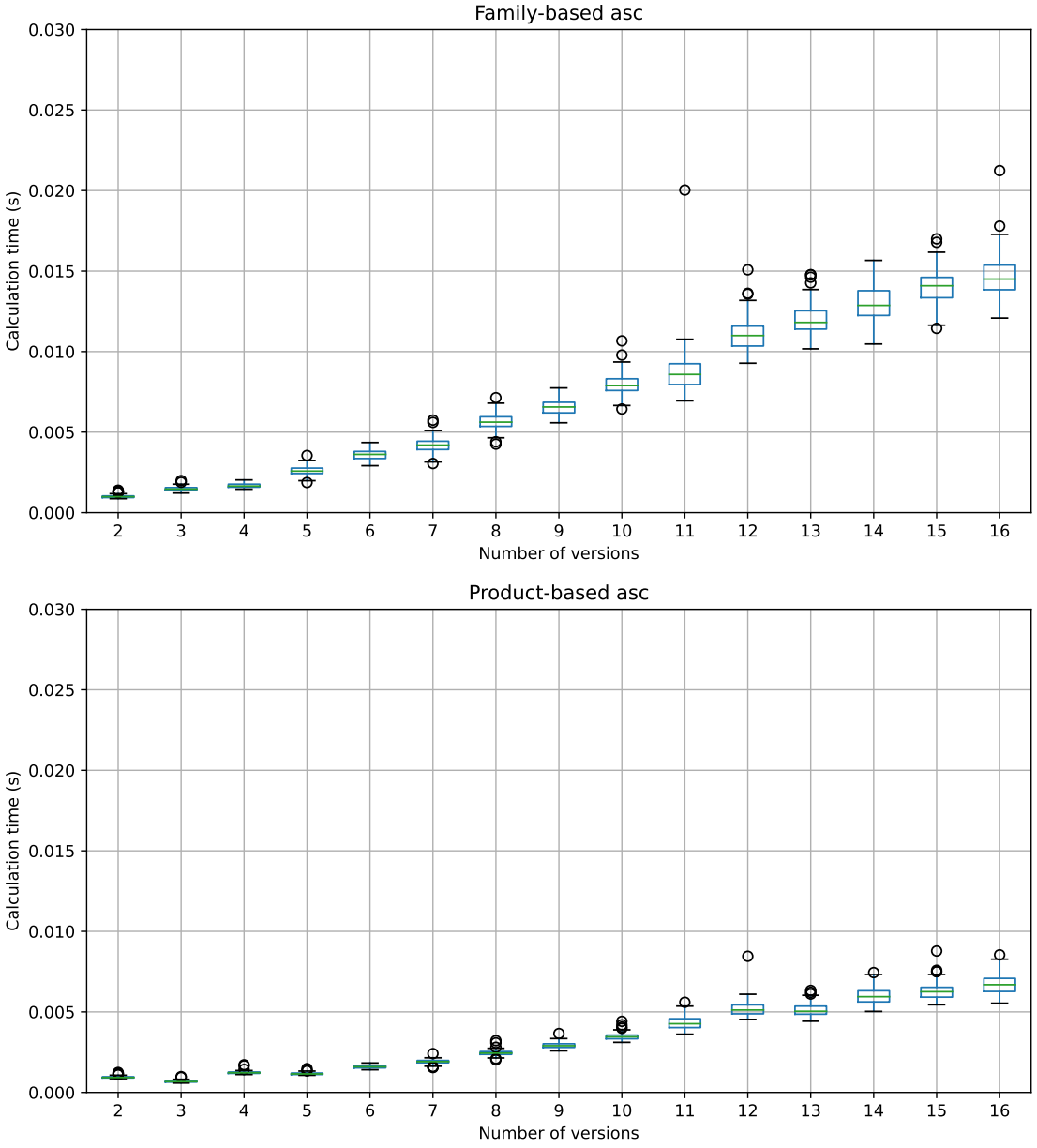


Figure 5.1: Execution time comparison: family-based vs. product-based

EQ1: family-based Online Machine Separation vs product-based

How does the family-based Online Machine Separation perform against the product-based version?

In terms of execution time, the product-based Online Machine Separation performs better. However, the family-based highest execution time is around 21 milliseconds. Therefore, this is still a method which can be used in practice.

5.2 CADS vs. CPDS

In this section, we verify if the Configuration-based Adaptive Distinguishing Sequence (CADS) is better than the Configuration-based Preset Distinguishing Sequence (CPDS). In regular distinguishing sequences, an ADS exists more often and is shorter in most cases than a PDS. Therefore we verify if this holds for the configuration versions. By comparing these two methods, we answer EQ2.

5.2.1 Methodology

To verify if the CADS is the most efficient, we measure and compare them based on the time taken to compute the CADS/CPDS and the efficiency of the CADS/CPDS for the 16 distinct OpenSSL versions. Just as for the Online Machine Separation comparison, we run the experiments with the minimum available models (2) up to the 16 distinct versions in ascending order. The difference is here that for the execution time, we have set a time limit of 20 minutes. When the computation takes longer than this time limit, we stop the calculation of the CDS. Also, every option is executed ten times.

To compare the execution time results, we first perform statistical hypothesis testing using the Mann-Whitney U to determine statistical significance. If we obtain statistical significance, we calculate the effect size using Vargha and Delaney's metric [3]. For comparing the efficiency of the sequences, we look at the number of inputs and resets. If one sequence has more inputs than the other, it takes more time and effort to detect a configuration. Second, a reset is not optimal as it takes some time for the system to be back in operational mode. Therefore we want to have as minimum resets as possible. To compare the number of inputs and resets, we compare the average and maximum numbers that are needed to detect a version. Here the average is most interesting for the CADS approach as not every version needs the same number of inputs to distinguish them. With an average, it is hard to determine if we have cases where the number of inputs and resets is large. Therefore we also consider the worst case of the inputs and resets by comparing the maximum number of inputs and resets.

For all the results, we create a box plot and a table containing the mean, standard deviation, minimum and maximum value. The plots will be shown in this sections while the tables are listed in Appendix C.

5.2.2 Results

As discussed in the methodology, we compare the CADS and CPDS on the execution time it takes to compute the sequences and the efficiency of the resulting sequence. In the first section, we compare them on the execution time, and in the second section we compare them on the efficiency of the sequence.

5.2.2.1 Execution time comparison

In the execution time comparison of the CADS and CPDS approaches, we see that the CADS approach is faster for larger family model representations (i.e. more versions). In Table 5.2, we see that for all cases the p-value is less than 0.05 which means that we have statistical significance. Now, for sizes 2 to 8 and 10, the CPDS approach has an effect size very close to 1, indicating a very large effect. This means that the execution time of the CADS approach is higher in almost all cases. Therefore on these sizes, we conclude that the CPDS approach is faster. However, when the models are getting bigger (from 11 versions and on), we see that the CADS approach is faster.

Nr of versions	p-value (Mann-Whitney U)	Effect size CPDS vs. CADS	Magnitude	Avg. time (s) CPDS	Avg. time (s) CADS
2	0.000183	1	Large	0.001166	0.012065
3	0.000183	1	Large	0.002343	0.031932
4	0.000183	1	Large	0.002453	0.042078
5	0.000183	1	Large	0.004452	0.065908
6	0.000183	1	Large	0.022725	0.121627
7	0.000183	1	Large	0.038360	0.174496
8	0.001008	0.94	Large	0.233280	0.299723
9	0.000183	0	Large	1.077473	0.524601
10	0.000183	1	Large	4.024490	7.728839
11	0.001706	0.08	Large	20.708756	16.689287
12	0.000183	0	Large	59.191610	18.882142
13	0.000183	0	Large	75.443015	26.488269
14	0.000183	0	Large	321.700199	31.420006
15	0.000111	0	Large	1132.775543	49.832779
16	0.000149	0	Large	1123.779180	49.375182

Table 5.2: Execution time comparison: CPDS vs. CADS

When we look at the box plot in Figure 5.2 or Table C.1, we see that for every execution, the CADS approach finishes within one minute. For the CPDS we have 14 executions which reached the time limit. From these 14 executions, eight reached the time limit with 15 versions, and it happened six times with 16 versions. Also, we see that on average for the versions where the CPDS is faster, the CPDS is maximal around 4 seconds faster than the CADS. However, if we consider it the other way around, we see that on average the CADS is maximal around 18 minutes faster than the CPDS. From this, we conclude that in some cases the CPDS is faster than the CADS. But when it comes to practical usage, the CADS approach has the execution time to its advantage by finishing all cases within one minute and is thus the better option.

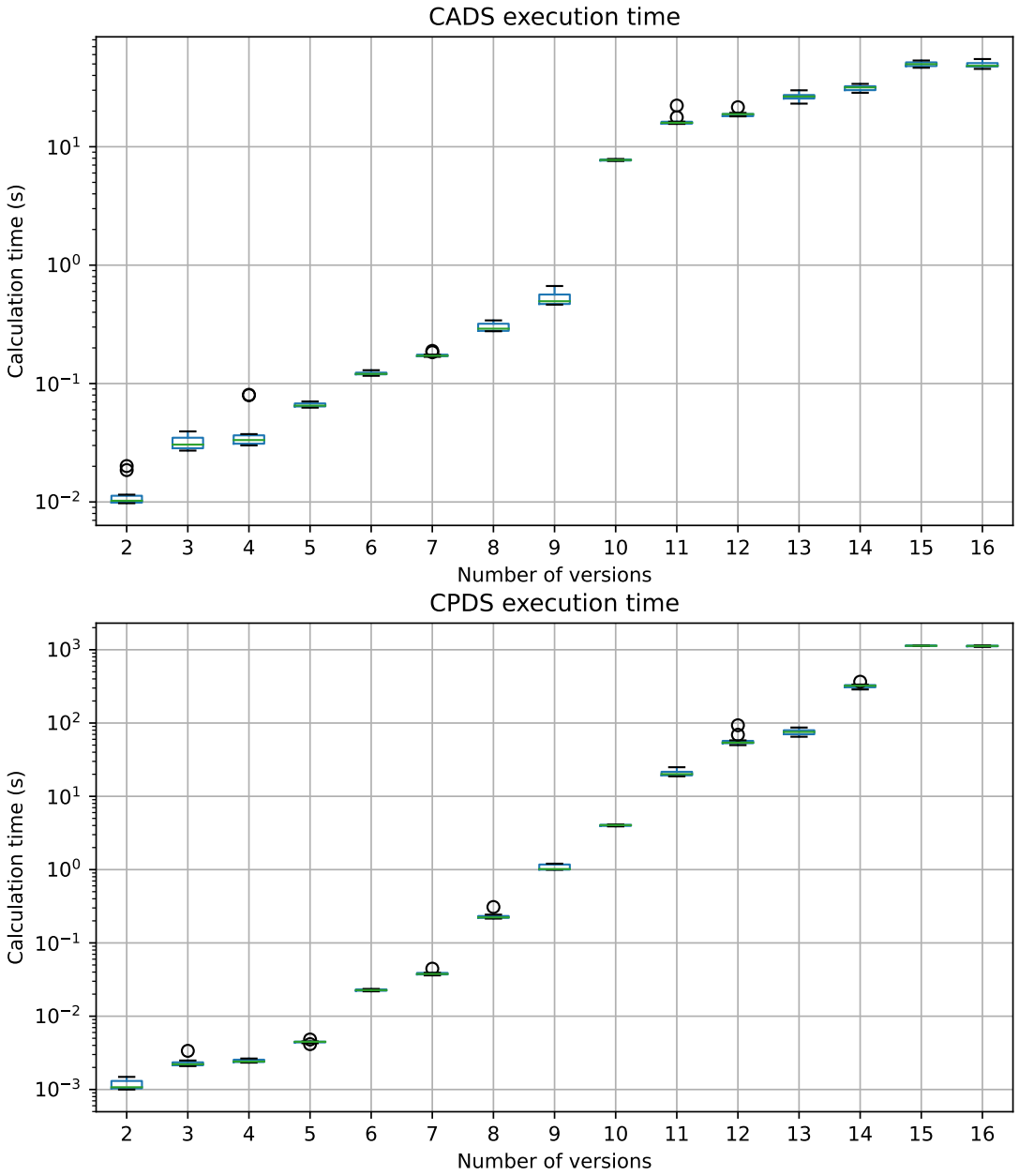


Figure 5.2: Box plot time comparison: CADS vs. CPDS

5.2.2.2 Efficiency comparison

To compare the efficiency of the calculated sequence, we compare the number of inputs and resets of the sequence. At first, we compare them on the number of inputs in the sequence. With Figure 5.3 or Table C.4, we consider the average inputs to distinguish a version. In these metrics, we see that only for the case of two versions, the CPDS and CADS need the same amount of inputs to detect a version. For all other cases, the CADS requires fewer inputs on average. When we consider the maximum number of inputs from Figure 5.4 or Table C.3, we see that from eight versions and on, the CADS on average needs fewer inputs at its maximum. But there are also some exceptions, for example at 12 and 13 versions, where the CADS can calculate a sequence which is longer than the CPDS. Since we see that the CADS is more efficient in most cases on both average and maximum values, we conclude that the CADS is more efficient in terms of inputs.

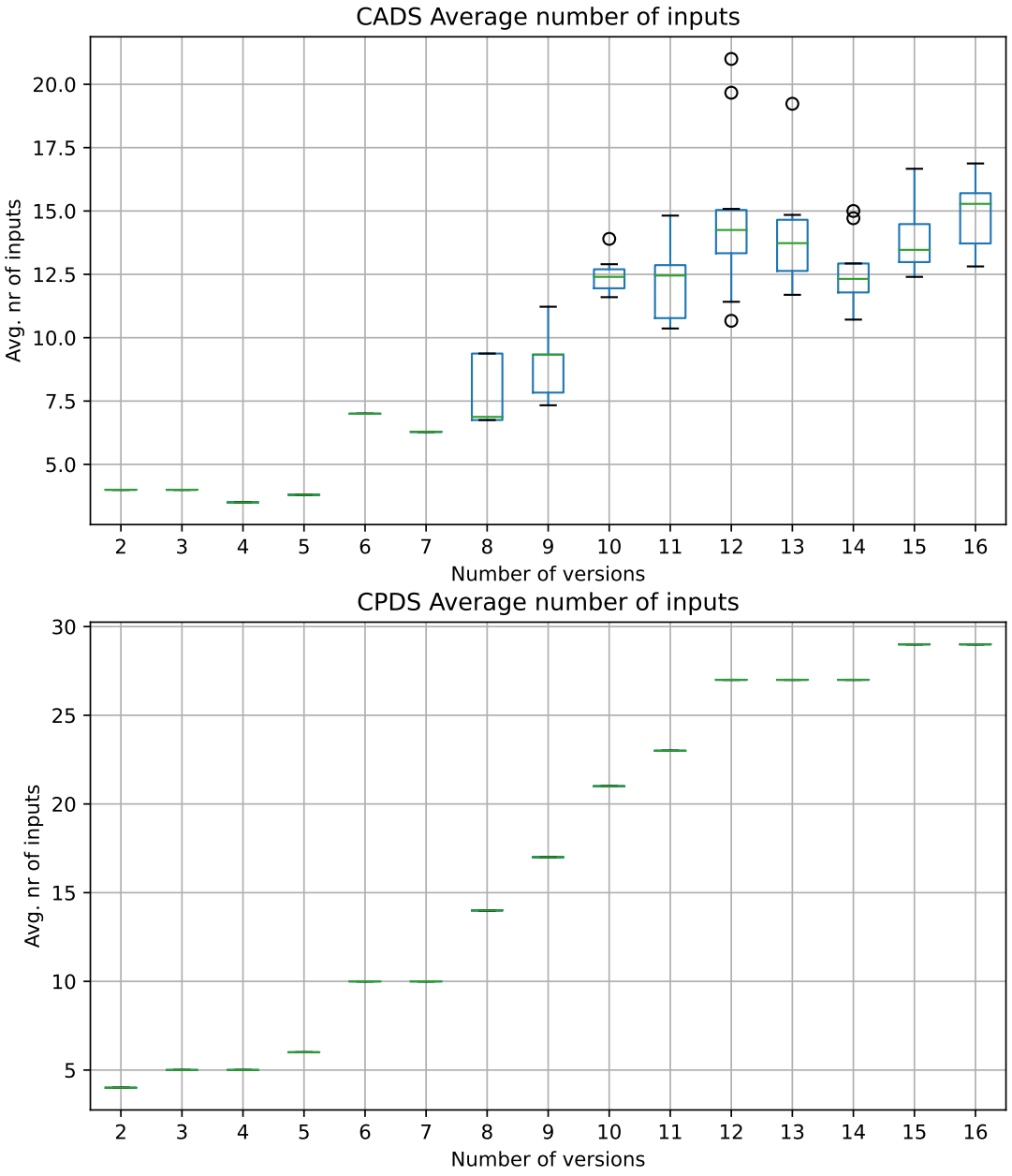


Figure 5.3: Average number of inputs: CADS vs. CPDS

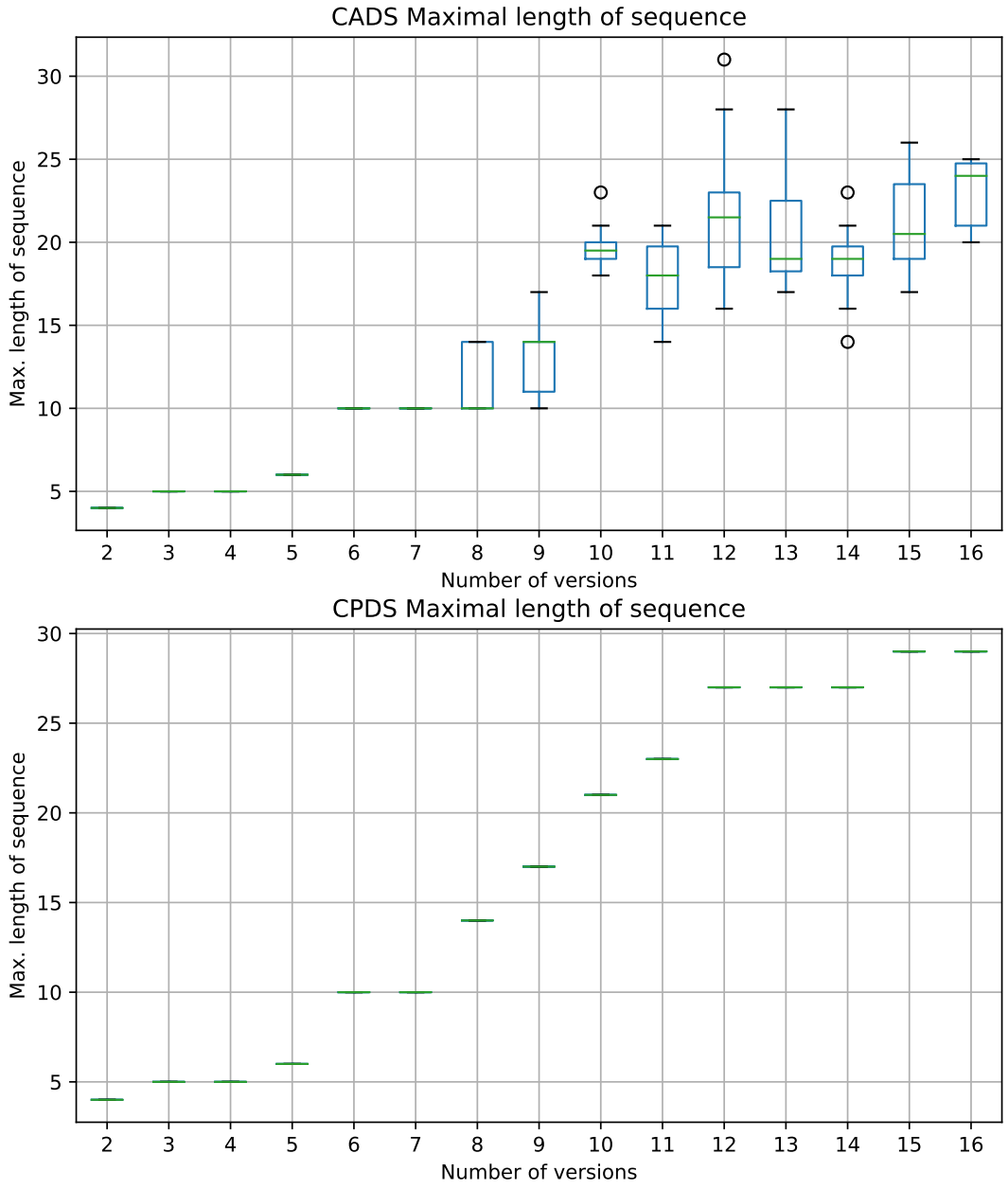


Figure 5.4: Maximal number of inputs: CADS vs. CPDS

Second, we compare the CADS and CPDS on the number of resets required to distinguish a version. When comparing the average resets from Figure 5.5 or Table C.4, we see that the CADS needs fewer resets on average from six versions and on. When we compare the maximal number of resets from Figure 5.6 or Table C.5, we see that difference is smaller. from eight versions on, the CADS approach often applies fewer resets than the CPDS. But for 12 versions, there exists a case where the CADS approach applies more resets than the CPDS. But in most cases, the CADS approach requires fewer resets to distinguish a version. Therefore we conclude that the CADS approach is more efficient than the CPDS in terms of resets.

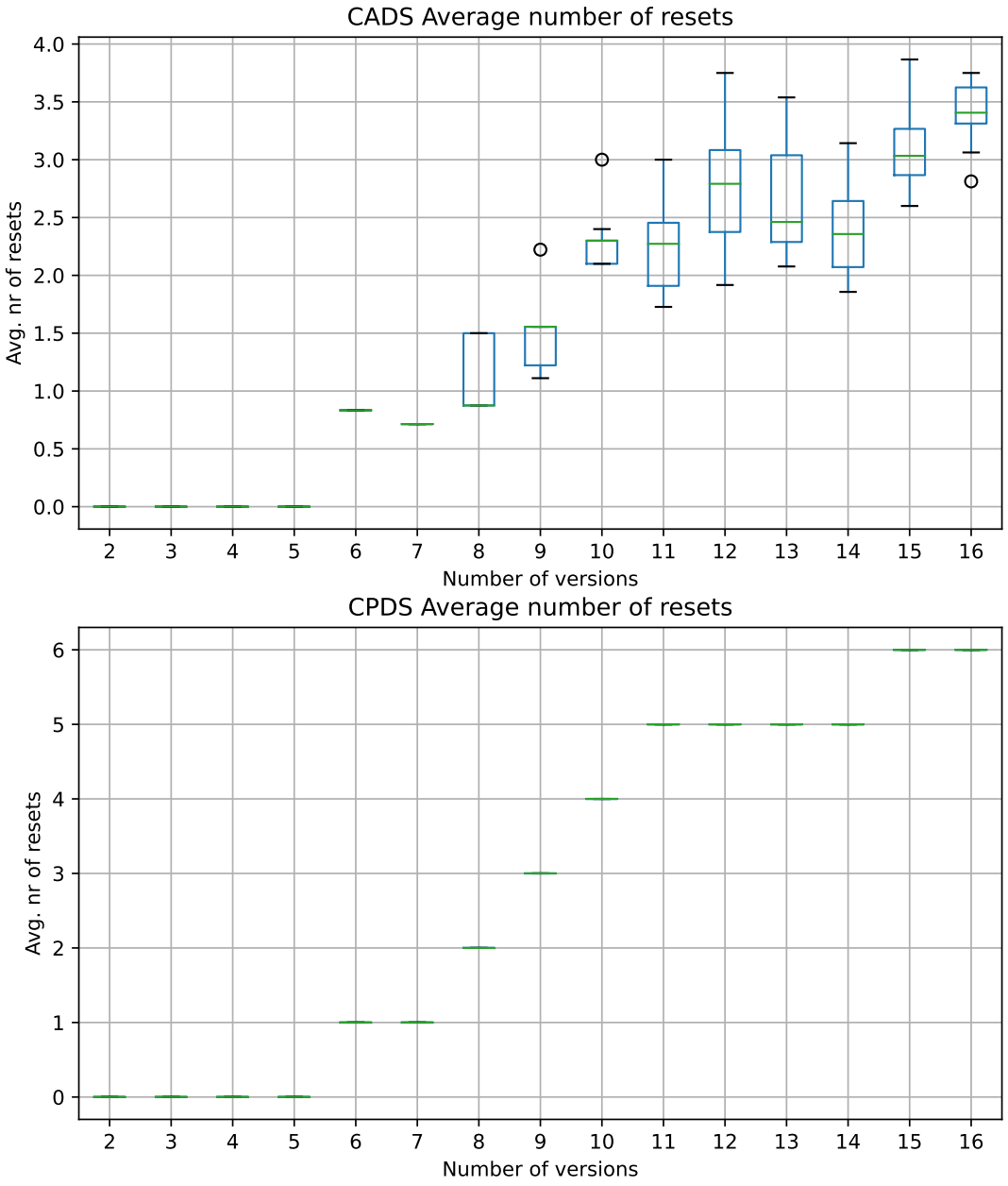


Figure 5.5: Average number of resets: CADS vs. CPDS

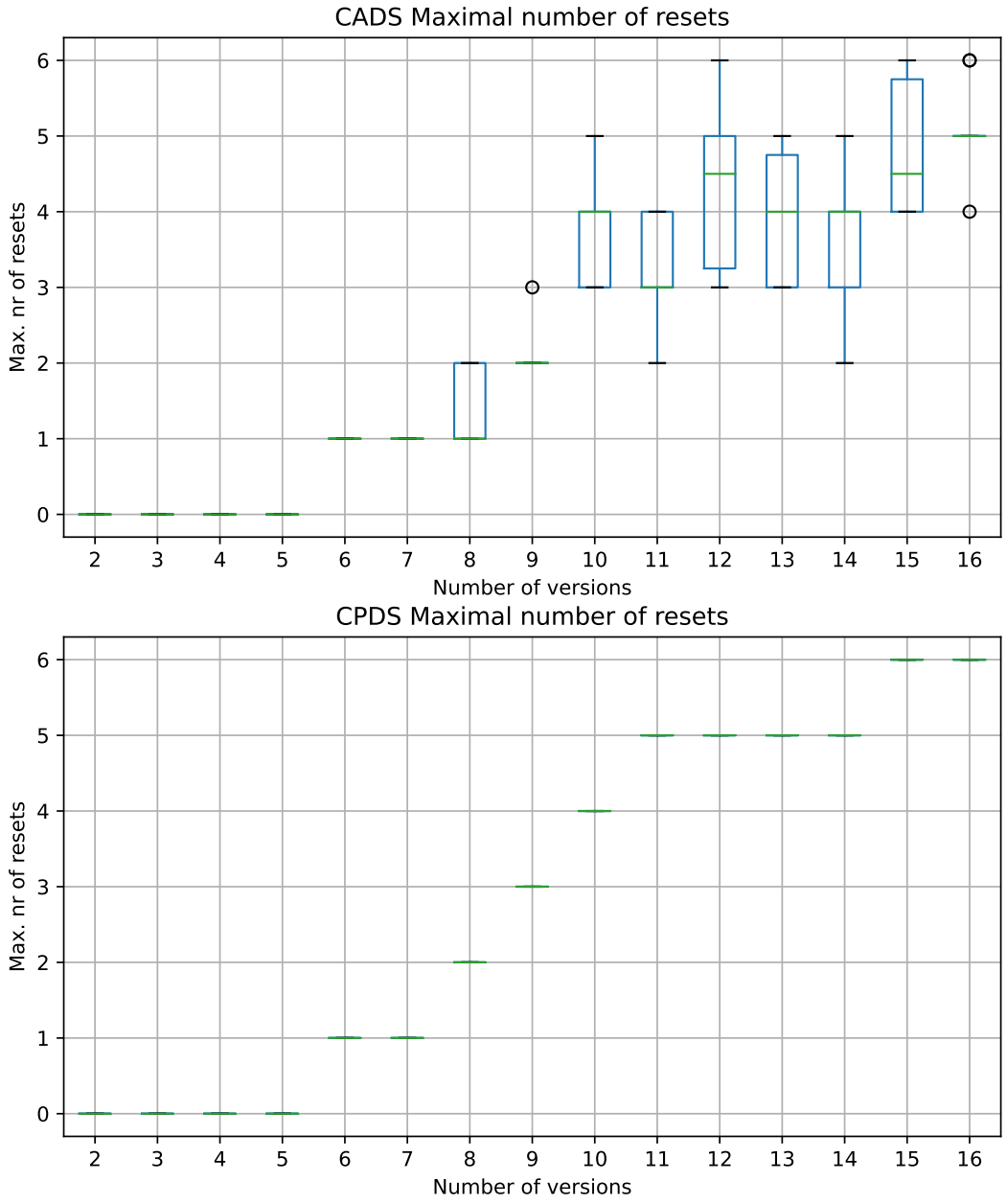


Figure 5.6: Maximal number of resets: CADS vs. CPDS

EQ2: CADS vs. CPDS

How much more efficient is the Configuration-based Adaptive Distinguishing Sequence over the Preset?

The adaptive approach is more efficient than the preset approach. In terms of sequence efficiency, the adaptive solution calculates a sequence which needs to apply fewer inputs and resets. In terms of execution time, the adaptive approach finishes one run in a maximum of one minute, while the preset had a total of 14 time-outs.

5.3 Online Machine Separation vs. CADS

As seen in Section 5.2, the CADS outperforms the CPDS approach, we now investigate how CADS approach performs against the family-based Online Machine Separation approach. This answers our formulated EQ3.

5.3.1 Methodology

To test which of the two methods is the best, we compare the approaches in the same way as the CPDS against the CADS in Section 5.2. The only difference is that in this case we do not have a time limit and run every option 50 times instead of 10.

The tables for this comparison are listed in Appendix D.

5.3.2 Results

In this section, we discuss the results we found. First, we discuss the execution time and the efficiency of the sequences.

5.3.2.1 Execution time comparison

When comparing the family-based version of the Online Machine Separation approach and the CADS approach, we first look at the p-value in Table 5.3 and see that all numbers are smaller than 0.05 and we thus obtained statistical significance. When we now look at the effect size, we see that the effect size is always one, indicating a very large effect. Also, this means that in all cases the CADS was slower than the Online Machine Separation. If we look at Figure 5.7 or Table D.1, we see that on average, the Online Machine Separation approach can solve the 16 versions in around 16 milliseconds. While for the CADS approach, the average time is around 46 seconds. Since the effect size is always 1 and the Online Machine Separation approach has a lower execution time than the CADS approach, we conclude that the Online Machine Separation approach is more efficient in terms of execution time.

Nr of versions	p-value (Mann-Whitney U)	Effect size OMS vs. CADS	Magnitude	Avg. time (s) OMS	Avg. time (s) CADS
2	0	1	Large	0.001106	0.010142
3	0	1	Large	0.001621	0.028200
4	0	1	Large	0.001773	0.031219
5	0	1	Large	0.003208	0.071430
6	0	1	Large	0.003916	0.125340
7	0	1	Large	0.004583	0.180602
8	0	1	Large	0.006120	0.275155
9	0	1	Large	0.007039	0.499031
10	0	1	Large	0.008470	7.916353
11	0	1	Large	0.009265	16.023204
12	0	1	Large	0.011587	16.023204
13	0	1	Large	0.012143	21.873293
14	0	1	Large	0.013751	27.989035
15	0	1	Large	0.014497	44.526636
16	0	1	Large	0.015548	45.954853

Table 5.3: Execution time comparison: CADS vs. Online Machine Separation

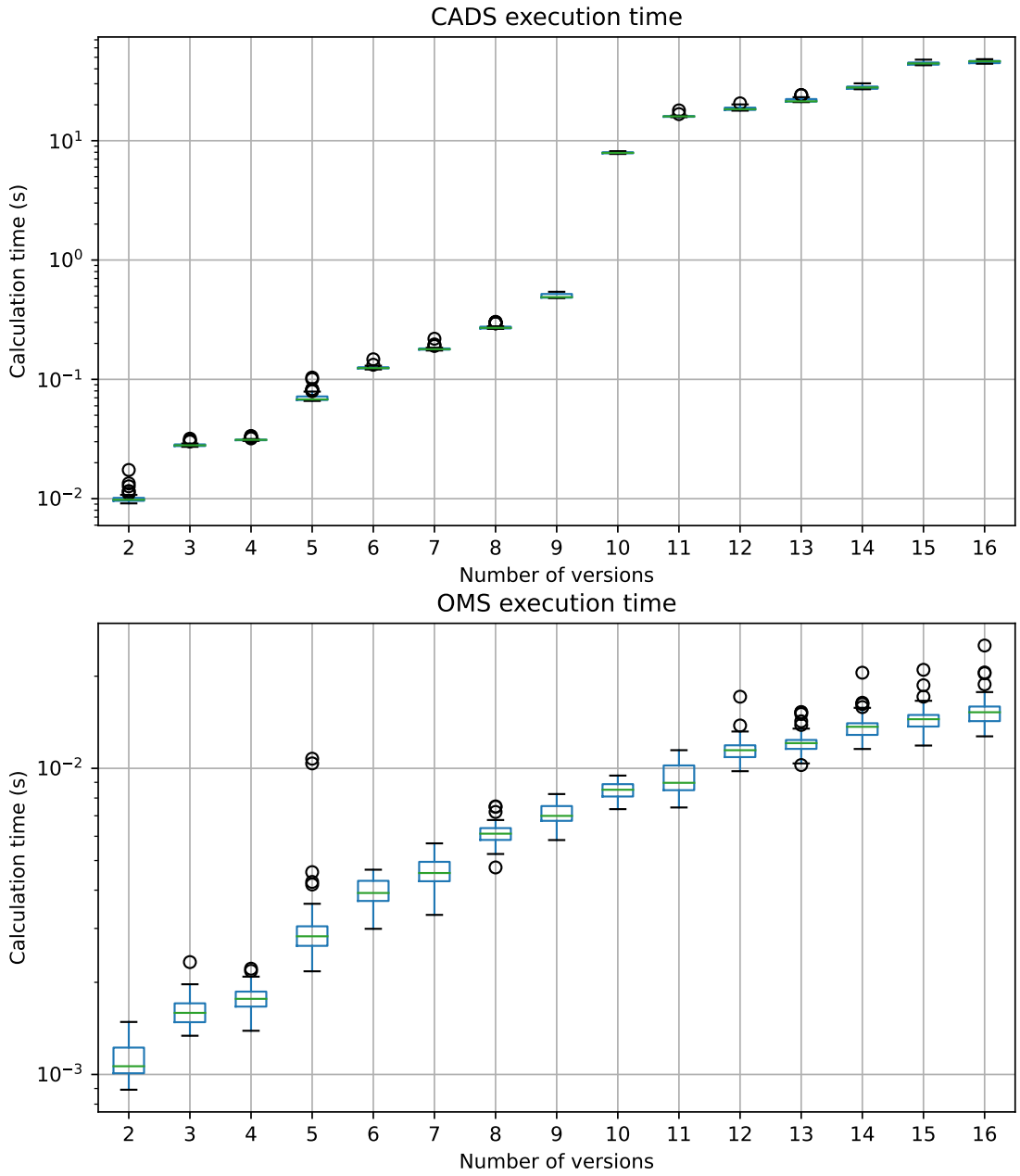


Figure 5.7: Execution time comparison: CADS vs. Online Machine Separation

5.3.2.2 Efficiency comparison

To compare the efficiency of the calculated sequences, we compare the number of inputs and resets of the sequence. At first, we compare them on the number of inputs in the sequences. For the inputs, we consider the average and maximal number of inputs we apply to the system to detect a version. When we consider the average number of inputs from Figure 5.8 or Table D.2, we see only for two versions that the Online Machine Separation and CADS need the same number of inputs on average. But whenever we add more versions, the CADS approach needs fewer inputs. When we consider the maximal number of inputs that are needed to distinguish a version, the results are quite similar. In Figure 5.9 or Table D.3, we see that for two versions the same number of inputs are required. However, when we distinguish more versions, the CADS has a smaller

maximal sequence in almost all cases. What is also remarkable, is the fact that the maximal length is equal to the average for the Online Machine Separation approach while for the CADS the average is smaller than the maximum. Therefore we conclude that in terms of inputs, the CADS is more efficient.

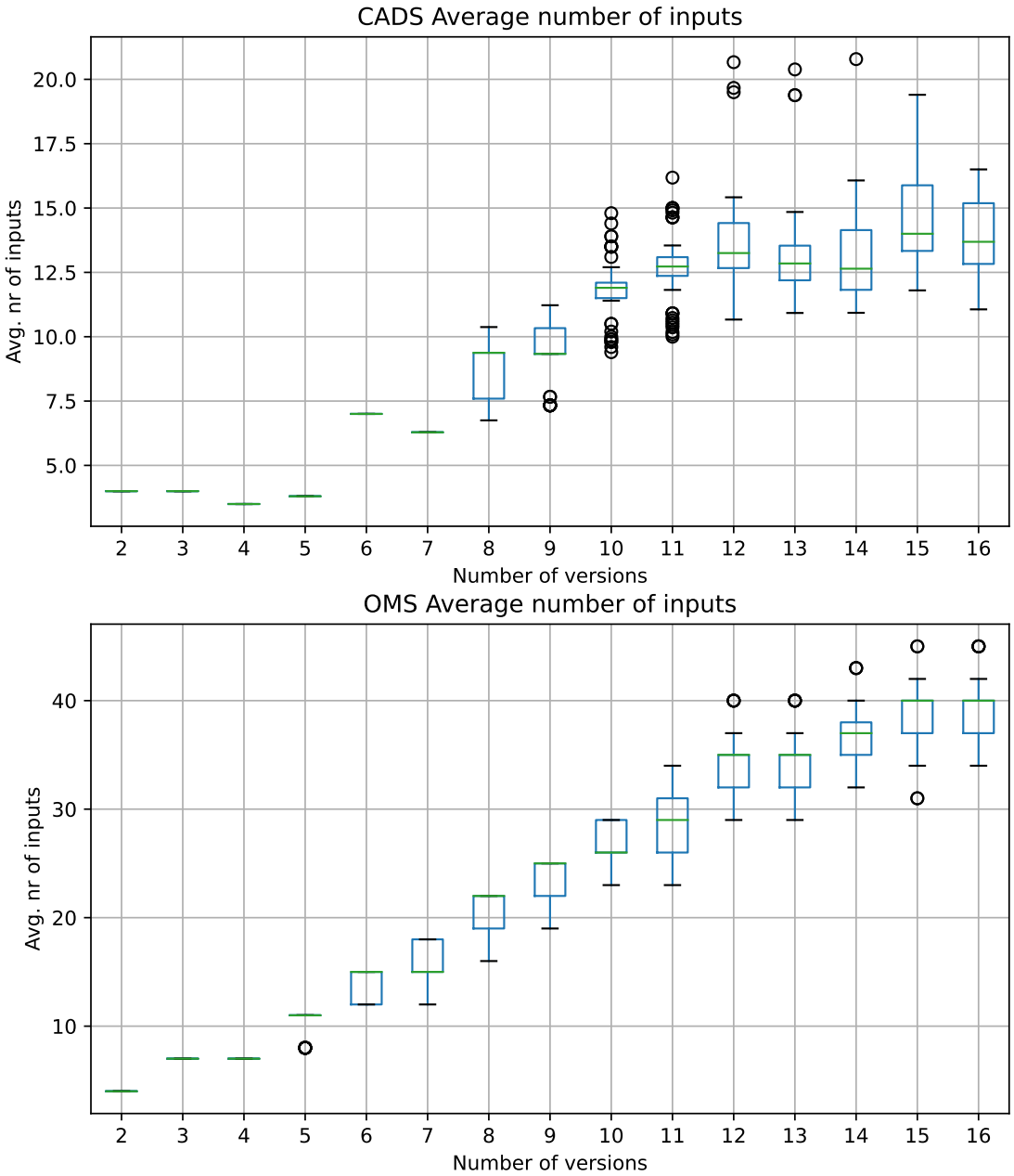


Figure 5.8: Average number of inputs: CADS vs. Online Machine Separation

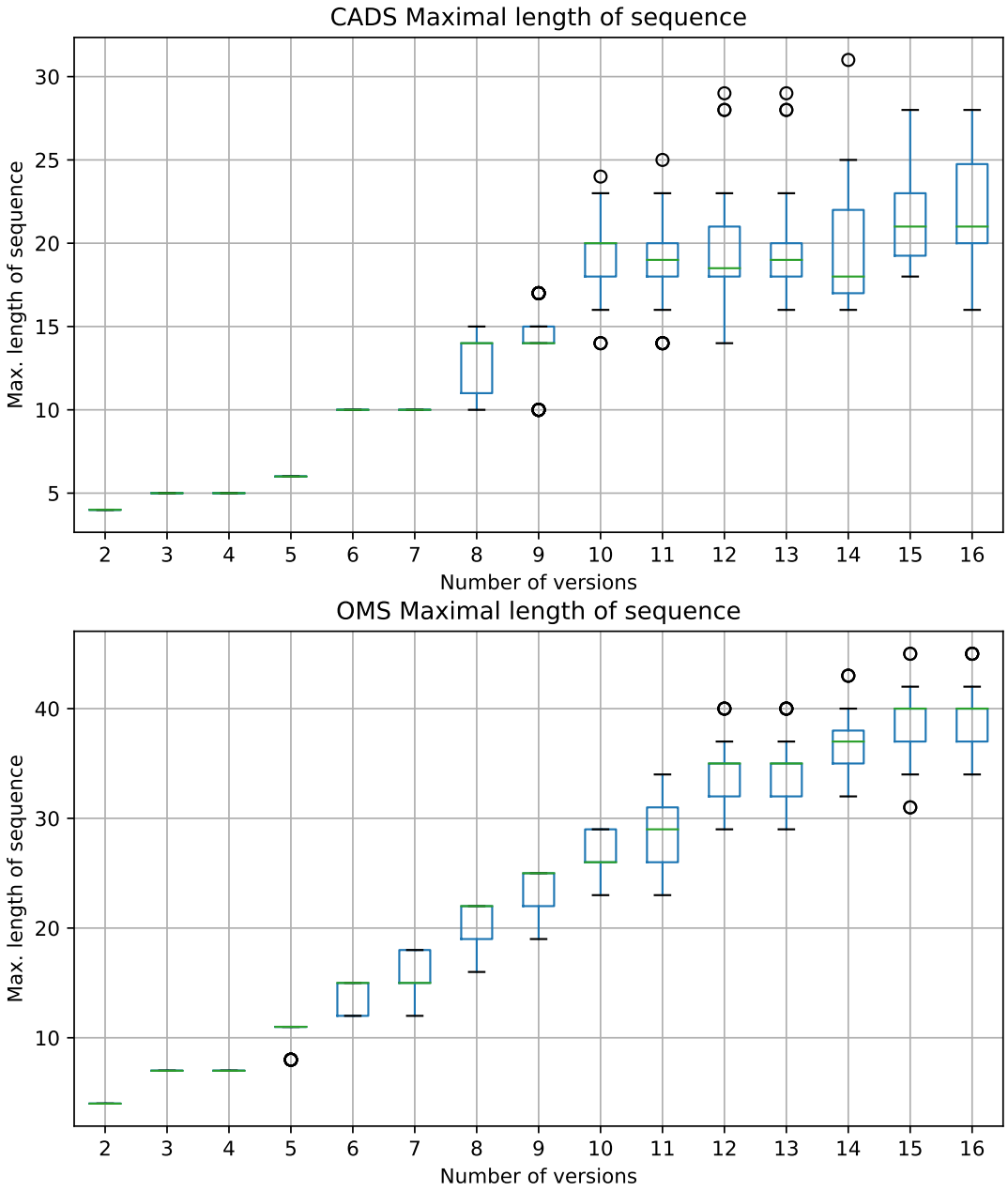


Figure 5.9: Maximal number of inputs: CADS vs. Online Machine Separation

Second, we compare the required number of resets to distinguish versions. At first, we consider the average number of resets of the sequence, we see in Figure 5.10 or Table D.4 that only for two versions the number of resets is equal. For all the other cases, the CADS approach requires fewer resets to distinguish versions. Also, the CADS needs to reset the system from six versions while the Online Machine Separation requires this from three versions. Second, we consider the maximal number of resets of the sequence. In Figure 5.11 or Table D.5, we see that the average maximum of the CADS approach uses fewer resets. Even the highest maximum number of resets from the CADS is always less or equal to the lowest maximum of the Online Machine Separation approach. Therefore we conclude that in terms of resets, the CADS is more efficient.

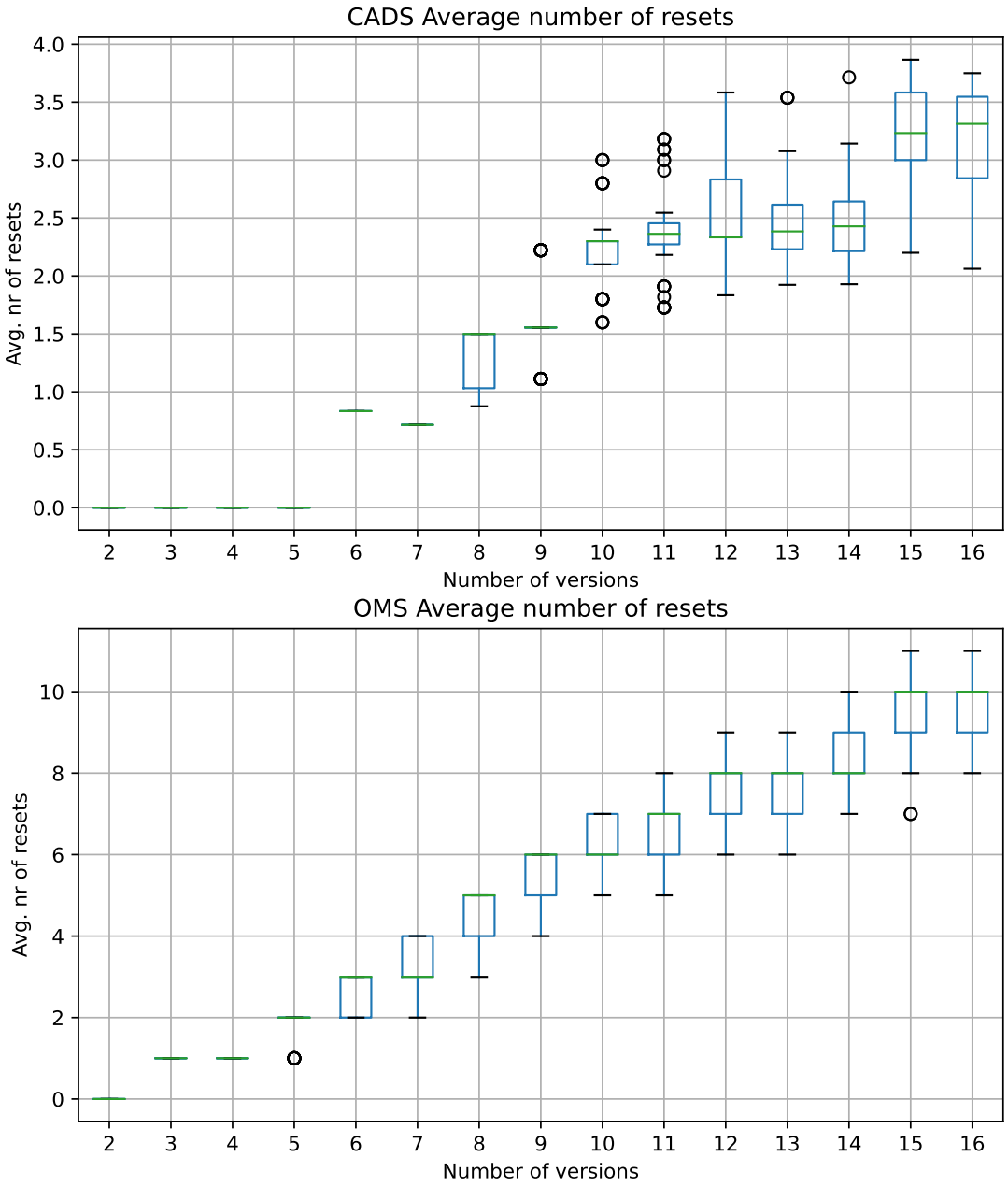


Figure 5.10: Average number of resets: CADS vs. Online Machine Separation

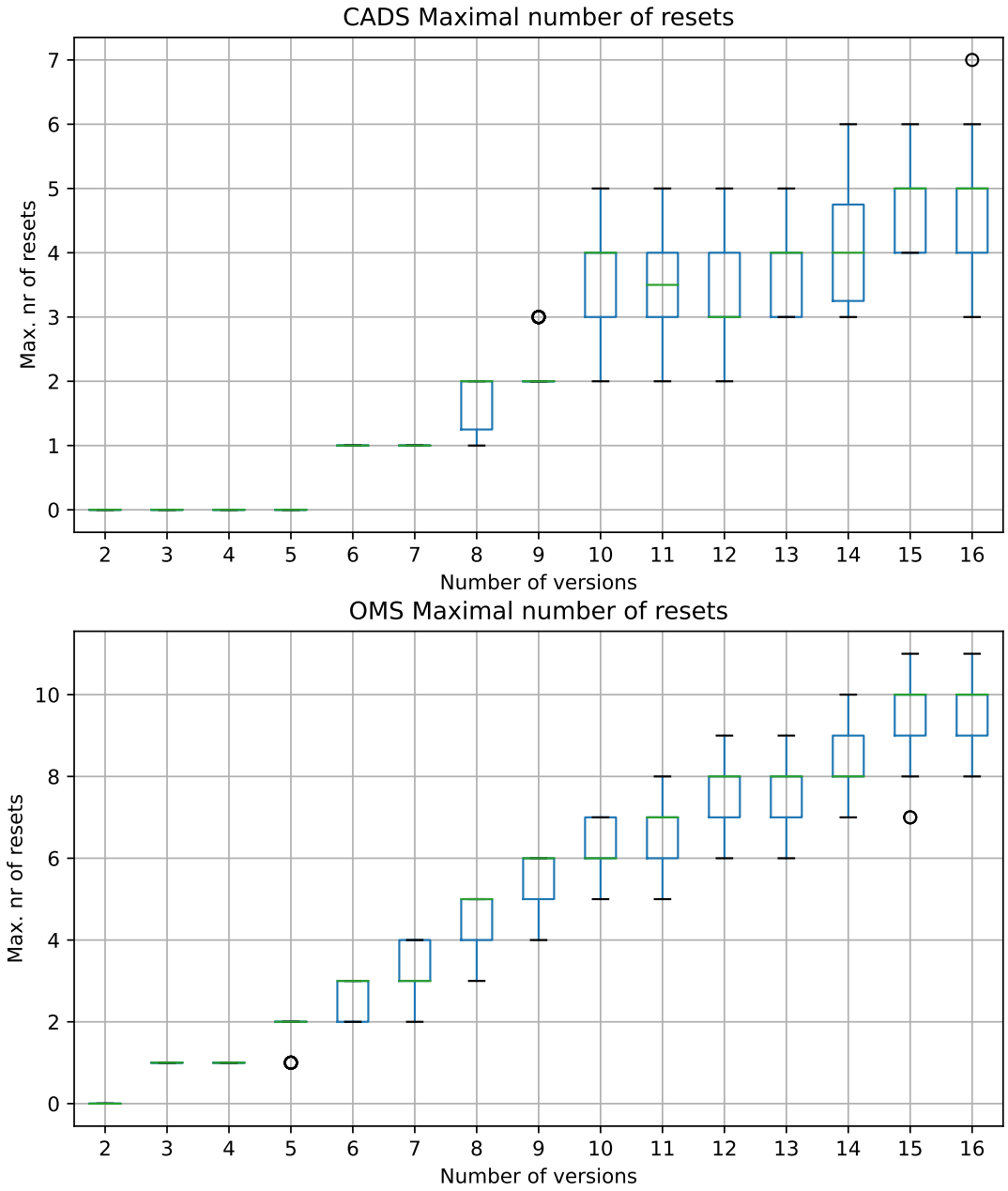


Figure 5.11: Maximal number of resets: CADS vs. Online Machine Separation

EQ3: Online Machine Separation vs. CADS

How does family-based Online Machine Separation perform against the Configuration-based Adaptive Distinguishing Sequence (CADS)?

In terms of execution time, the family-based Online Machine Separation is much faster than the Configuration-based Adaptive Distinguishing Sequence. While for the efficiency of the calculated sequences, the Configuration-based Adaptive Distinguishing Sequence is much better. Since the calculation of the sequences with the adaptive approach is still feasible (less than one minute), this approach is considered to be better for practical usage.

Chapter 6

Conclusion

In modern systems, security is one of the most critical requirements, but still there are many security issues in systems, in particular due to the widespread use of open-source project and variability-intensive software, for which security is particularly hard to establish. Unfortunately, it is often hard to detect if a system is running a vulnerable version. A solution for this is software fingerprinting, where one can discover which version of a system is running in the black-box system. In Chapter 2, we discussed the existing fingerprinting approaches, which use a set of candidates represented by PEFSM models to calculate a set of sequences which separates the models. Also, we discussed methods of distinguishing different states within an FSM model.

In this thesis, we lift the notion of fingerprinting product families from a product-based approach to a family-based approach. For this we use a family-based signature, the FFSM representation. This representation is more efficient as the commonalities and variability's of the products are described. Current product families do not always have our family-based representation available. Therefore, we proposed a way of creating the model in Section 4.1 by representing the different available products as a single feature in the feature model. This enables a lightweight process to analyse the variability for the different versions.

As there was no other work on family-based fingerprinting, we first examined the product-based approaches discussed by Shu and Lee and lifted these notions to our family-based representation. Here in Section 4.2 and Section 4.3.2, we discuss the changes that are required to the algorithms to work with our FFSM representation. Also, we came to the conclusion that this family-based Online Machine Separation has to calculate $\mathcal{O}(\Lambda^2)$ sequences, which is not ideal.

Therefore, to compute more efficient sequences, we wanted to apply the algorithms from the theory of state distinguishing sequences. Since we distinguish different configurations, these algorithms do not work perfectly. As described in Section 4.3.3, with some adaptations to the existing algorithm, it was possible to find a preset distinguishing sequence for configurations. While for the adaptive distinguishing sequence, there did not exist an algorithm which could be adapted to work. Therefore, we proposed a new algorithm which finds an adaptive distinguishing sequence for configurations in Section 4.3.4.

In Chapter 5, we compared the different active fingerprints in terms of the efficiency of the sequences and the efficiency of calculating the sequences. Instead of comparing them all at once, we compared the most interesting ones in pairs. This means that the product-based Online Machine Separation is compared to the family-based Online Machine Separation, the preset is compared to the adaptive sequences, and finally, the family-based Online Machine Separation is compared to the adaptive implementation.

In terms of calculating efficiency, we found that our family-based Online Machine Separation calculates the sequence slower than the product-based method but much faster than the preset and adaptive approaches. However, when we look at the absolute value difference between the Online Machine Separation approaches, we see that this is on average maximal ten milliseconds between the two fingerprinters. So although the product-based method is faster, the family-based implementation can still be considered a good alternative with the benefit of a family-based representation.

In terms of sequence efficiency, the CPDS and CADS both can calculate sequences which have fewer resets and inputs than the Online Machine Separation approach. From these two better options, the CADS applies the fewest amount of inputs and resets. Compared to the Online Machine Separation approach, the CADS reduces the average number of inputs required to distinguish one configuration by 25 and the average number of resets by 6.

All algorithms and benchmarks are made public in the GitHub repository[1]. In Appendix A, we discuss the different folders and how to run the benchmarks and fingerprinters.

We now discuss potential directions for future work. One direction is to develop new methods using the family-based fingerprinting. Janssen [15] showed that it is possible to fingerprint systems based on a heuristic decision tree. This method takes a Finite State Machine and normalizes this model to a tree. In the next step, the created trees are combined into one big heuristic decision tree. But we already have a unified representation, namely the FFSM. Therefore it would make sense to derive this heuristic decision tree directly from the FFSM. In this way, it is not necessary to combine the trees afterwards to create the heuristic decision trees and we have the benefit of a more compact signature as a representation of the different systems. Also, Janssen discussed the adaptive distinguishing graph (ADG). This calculated graph is very similar to our CADS and he stated the following: "Compute the ADG from the JSON files (adg-finder). For our combined models this takes around 10 to 15 minutes on our build server" [15]. Our experiments, which are not executed on the same build server, cost around one minute to find the CADS in the worst case. In future work, we can benchmark these two methods against each other. They can be compared on the efficiency of the sequence/graph and on the time it takes to compute the sequence/graph.

One more future work idea comes from the fact that we treat every version as a single feature. This gave us the possibility to use set theory for solving the feature constraints. When an FFSM is derived from a software product line, we do not have the different products as the presence conditions, but we have a feature representation as the presence conditions. It is then possible to generate all the distinct products from a product line and create a new FFSM with all the different products. But with the knowledge of optional features, this can lead to a combinatorial explosion of the actual set of products [7]. Therefore, a new strategy of family-based fingerprinting can be developed where we use the FFSM to derive the feature configuration of a black-box system by calculating a sequence that reveals which features are selected inside the black-box system. By using a SAT-solver we can verify if we have found a feature configuration that leads to a valid configuration. Furthermore, it is possible to set a constraint in the feature model for equivalent behavioural products. This solves one of the limitations we have with the current approaches.

[1] `https://github.com/GianniM123/FFSM_fingerprint`

Bibliography

[1] Shaukat Ali et al. "A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation". In: *IEEE Transactions on Software Engineering* 36.6 (2010), pp. 742–762. DOI: 10.1109/TSE.2009.52.

[2] Saed Alrabaee, Mourad Debbabi, and Lingyu Wang. "A Survey of Binary Code Fingerprinting Approaches: Taxonomy, Methodologies, and Features". In: *ACM Comput. Surv.* 55.1 (Jan. 2022). ISSN: 0360-0300. DOI: 10.1145/3486860. URL: https://doi.org/10.1145/3486860.

[3] Andrea Arcuri and Lionel Briand. "A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering". In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. Waikiki, Honolulu, HI, USA: Association for Computing Machinery, 2011, pp. 1–10. ISBN: 9781450304450. DOI: 10.1145/1985793.1985795. URL: https://doi.org/10.1145/1985793.1985795.

[4] Andreas Classen et al. "Model checking lots of systems: efficient verification of temporal properties in software product lines". In: *2010 ACM/IEEE 32nd International Conference on Software Engineering*. Vol. 1. 2010, pp. 335–344. DOI: 10.1145/1806799.1806850.

[5] Rance Cleaveland and Matthew Hennessy. "Testing equivalence as a bisimulation equivalence". In: *Formal Aspects of Computing* 5 (1993), p. 20. ISSN: 1433-299X. DOI: 10.1007/BF01211314. URL: https://doi.org/10.1007/BF01211314.

[6] Maxime Cordy et al. "A Decade of Featured Transition Systems". In: *From Software Engineering to Formal Methods and Tools, and Back: Essays Dedicated to Stefania Gnesi on the Occasion of Her 65th Birthday*. Ed. by Maurice H. ter Beek, Alessandro Fantechi, and Laura Semini. Cham: Springer International Publishing, 2019, pp. 285–312. ISBN: 978-3-030-30985-5. DOI: 10.1007/978-3-030-30985-5_18. URL: https://doi.org/10.1007/978-3-030-30985-5_18.

[7] Carlos Diego N. Damasceno and Daniel Strüber. "Family-Based Fingerprint Analysis: A Position Paper". In: *A Journey from Process Algebra via Timed Automata to Model Learning : Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday*. Ed. by Nils Jansen, Mariëlle Stoelinga, and Petra van den Bos. Cham: Springer Nature Switzerland, 2022, pp. 137–150. ISBN: 978-3-031-15629-8. DOI: 10.1007/978-3-031-15629-8_8. URL: https://doi.org/10.1007/978-3-031-15629-8_8.

[8] Carlos Diego Nascimento Damasceno, Mohammad Reza Mousavi, and Adenilso da Silva Simao. "Learning by sampling: learning behavioral family models from software product lines". en. In: *Empirical Software Engineering* 26 (Jan. 2021), p. 4. ISSN: 1573-7616. DOI: 10.1007/s10664-020-09912-w. URL: https://doi.org/10.1007/s10664-020-09912-w.

[9] Rocco De Nicola and Roberto Segala. "A process algebraic view of input/output automata". In: *Theoretical Computer Science* 138.2 (1995). Meeting on the mathematical foundation of programing semantics, pp. 391–423. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(95)92307-J. URL: https://www.sciencedirect.com/science/article/pii/030439759592307J.

[10] Manuel Egele et al. “A Survey on Automated Dynamic Malware-Analysis Techniques and Tools”. In: *ACM Comput. Surv.* 44.2 (Mar. 2008). ISSN: 0360-0300. DOI: 10.1145/2089125.2089126. URL: https://doi.org/10.1145/2089125.2089126.

[11] Jérôme François et al. *Behavioral and Temporal Fingerprinting*. Intern report RR-6995. INRIA, 2009, p. 22. URL: https://hal.inria.fr/inria-00406482.

[12] Jérôme François et al. “Semi-supervised Fingerprinting of Protocol Messages”. In: *Computational Intelligence in Security for Information Systems 2010*. Ed. by Álvaro Herrero et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 107–115. ISBN: 978-3-642-16626-6.

[13] Arthur Gill. *Introduction to the Theory of Finite State Machines*. New York: McGraw-Hill, 1962.

[14] Vanderson Hafemann Fragal, Adenilso Simao, and Mohammad Reza Mousavi. “Validated Test Models for Software Product Lines: Featured Finite State Machines”. In: *Formal Aspects of Component Software*. Ed. by Olga Kouchnarenko and Ramtin Khosravi. Cham: Springer International Publishing, 2017, pp. 210–227. ISBN: 978-3-319-57666-4. DOI: 10.1007/978-3-319-57666-4_13. URL: https://doi.org/10.1007/978-3-319-57666-4_13.

[15] Erwin Janssen. “Fingerprinting TLS Implementations Using Model Learning”. en. [Online]. MSc dissertation. Nijmegen: Radboud University, Mar. 2021. (Visited on 09/19/2022).

[16] Andy Kenner et al. “Safety, Security, and Configurable Software Systems: A Systematic Mapping Study”. In: *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A*. SPLC ’21. Leicester, United Kingdom: Association for Computing Machinery, 2021, pp. 148–159. ISBN: 9781450384698. DOI: 10.1145/3461001.3471147. URL: https://doi.org/10.1145/3461001.3471147.

[17] Johannes Kinder. “Static Analysis of x86 Executables”. en. text. Technische Universität Darmstadt, Nov. 2010. URL: https://infoscience.epfl.ch/record/167546.

[18] Z. Kohavi. *Switching and Finite Automata Theory*. second edition. New York, NY: McGraw-Hill, 1978.

[19] Moez Krichen. “2 State Identification”. In: *Model-Based Testing of Reactive Systems: Advanced Lectures*. Ed. by Manfred Broy et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 35–67. ISBN: 978-3-540-32037-1. DOI: 10.1007/b137241. URL: https://doi.org/10.1007/b137241.

[20] D. Lee and M. Yannakakis. “Principles and methods of testing finite state machines-a survey”. In: *Proceedings of the IEEE* 84.8 (1996), pp. 1090–1123. DOI: 10.1109/5.533956.

[21] D. Lee and M. Yannakakis. “Testing finite-state machines: state identification and verification”. In: *IEEE Transactions on Computers* 43.3 (1994), pp. 306–320. DOI: 10.1109/12.272431. URL: https://ieeexplore.ieee.org/document/272431.

[22] Gianni Monteban. “Benchmarking SMT solvers and optimizing SMT within the LTSDiff algorithm”. en. In: (Jan. 2023). URL: https://github.com/GianniM123/ResearchInternship/blob/results/statistics/report.pdf.

[23] Sven Peldszus, Daniel Strüber, and Jan Jürjens. “Model-Based Security Analysis of Feature-Oriented Software Product Lines”. In: *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2018. Boston, MA, USA: Association for Computing Machinery, 2018, pp. 93–106. ISBN: 9781450360456. DOI: 10.1145/3278122.3278126. URL: https://doi.org/10.1145/3278122.3278126.

[24] Guoqiang Shu and David Lee. “A Formal Methodology for Network Protocol Fingerprinting”. In: *IEEE Transactions on Parallel and Distributed Systems* 22.11 (Nov. 2011), pp. 1813–1825. ISSN: 1558-2183. DOI: 10.1109/TPDS.2011.26. URL: https://ieeexplore.ieee.org/document/5686874.

[25] Jan Tretmans. "Model Based Testing with Labelled Transition Systems". In: *Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers*. Ed. by Robert M. Hierons, Jonathan P. Bowen, and Mark Harman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–38. ISBN: 978-3-540-78917-8. DOI: 10.1007/978-3-540-78917-8_1. URL: https://doi.org/10.1007/978-3-540-78917-8_1.

[26] Petra van den Bos and Frits Vaandrager. "State identification for labeled transition systems with inputs and outputs". In: *Science of Computer Programming* 209 (2021), p. 102678. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2021.102678. URL: https://www.sciencedirect.com/science/article/pii/S016764232100071X.

[27] Neil Walkinshaw and Kirill Bogdanov. "Automated Comparison of State-Based Software Models in Terms of Their Language and Structure". In: *ACM Trans. Softw. Eng. Methodol.* 22.2 (Mar. 2013). ISSN: 1049-331X. DOI: 10.1145/2430545.2430549. URL: https://doi.org/10.1145/2430545.2430549.

Appendices

Appendix A

Implementation details

In this chapter we discuss the details of the implemented algorithms. All the necessary code and information is found on GitHub[1] where different folders are created for the different topics. Below we give a detailed description per sub-folder of the GitHub.

A.1 FFSM_diff

This folder contains the implementation of the FFSM_diff algorithm. As a basis for the implementation, we took the implementation of the LTS_{diff} algorithm[2] and changed this implementation into the FFSM_{diff} algorithm. In the implementation, multiple options can be used to solve the linear equations. In earlier research [22], we concluded that Yices was the fastest in solving the equations. Therefore, we used and recommend to use this SMT solver to combine the models.

The FFSM_{diff} algorithm can be executed in the following manner:

```
python3 main.py
  --ref=<path/to/reference/model.dot>
  --upd=<path/to/updated/model.dot>
  --out=<path/to/output/model.dot>
  -s yices
```

A.2 Benchmarks

In this folder, the raw results of the benchmarks and the files to run the benchmarks can be found. By executing the *benchmark.py* with the option *all*, all the benchmarks performed in this research can be reproduced. To obtain the graphs and tables that are listed in the thesis, we feed one created *csv* to the *plot.py*. Below is shown how to run the scripts.

```
python3 benchmark.py all

python3 plot.py <comparison>.csv
```

A.3 Dot-files

This folder contains finite state machines of different systems or versions of one system in dot format.

[1] https://github.com/GianniM123/FFSM_fingerprint
[2] https://github.com/GianniM123/ResearchInternship

A.4 Equivalence-checker

Within this folder, an implementation is found to check if two FSMs are testing equivalent. The implementation takes two FSMs from a dot file and returns true if the models are equivalent and false otherwise. For this, we used Java and the automatalib[3]. Below is given how the equivalence checker can be executed.

```
mvn exec:java -D exec.mainClass=com.thesis.checker.App
-D exec.args="<path-to-ffsm.dot> <path-to-ffsm.dot>"
```

A.5 Family-based_fingerpinter

Within this folder, every implementation regarding the Family-based fingerprinter is made, this thus includes the family-based passive, Online Machine Separation, CPDS, and CADS fingerprinter. In the subsections, we discuss how these can be used.

A.5.1 Passive family-based fingerpinter

This fingerprinter is implemented in Python and to use it we first need to obtain an FFSM by using the FFSM_{diff} algorithm. When we have obtained the FFSM, we pick a trace that is observed from the system and put it inside a text file (Input/Output, Enter separated). Now we can run the passive fingerprinter in the following manner:

```
python3 main.py -f <path/to/ffsm.dot> -p <path/to/traces.txt>
```

A.5.2 Active family-based fingerprinters

These active fingerprinters are implemented in Python and to use them we first need to obtain an FFSM by using the FFSM_{diff} algorithm. When we have obtained the FFSM, we can select an FSM model which we want to fingerprint. Now, we can select a family-based fingerprinter to calculate the sequence. The options are *shulee*, *preset*, or *adaptive*. With the *shulee* option, the Online Machine Separation method is used, with *preset* the CPDS method is used, and with *adaptive* the CADS method is used.

```
python3 main.py -f <path/to/ffsm.dot> -a <path/to/fsm.dot> --option
```

Note that, by default, a *CDS.dot* is generated which contains the configuration-based distinguishing sequence. If you already have *CDS.dot* for a given FFSM, you can also use this sequence to detect a black-box system. This can be done in the following manner:

```
python3 main.py --sequence <path/to/CDS.dot> -a <path/to/fsm.dot> --option
```

A.6 Product-based_fingerpinter

In the product-based fingerprinter, we implemented the Online Machine Separation algorithm proposed by Shu and Lee and discussed in Section 2.3 in Python. To use the fingerprinter, we need to have a set of FSM models which are put in one folder. Next, we select one FSM which we want to fingerprint. Below, we can see how to run the product-based fingerprinter.

```
python3 main.py -f <path/to/ffsm.dot> -s <path/to/fsm.dot>
```

[3] https://learnlib.de/projects/automatalib/

A.7 Visualizer

The visualizer is used to visualize the feature representations of an FFSM. By default, the features are annotated within the dot files and not shown at the edges. With the visualizer, the feature representation is shown at the edges and nodes so it becomes easier to see which behaviour belongs to which feature. Below is shown how to run the visualizer, note that by default the output file is placed in the output folder.

```
python3 main.py <path/to/input-file.dot> <output-file.dot>
```

Appendix B

Online Machine Separation comparison

Nr of versions	p-value (Mann-Whitney U)	Effect size Product-based vs. Family-based	Magnitude	Avg. time (s) Product-based	Avg. time (s) Family-based
2	3.096e-31	0.9758	Large	0.000272	0.000513
3	1.542e-24	0.0815	Large	0.000860	0.000683
4	3.899e-34	0.9986	Large	0.000393	0.000767
5	2.186e-08	0.7291	Medium	0.001157	0.001360
6	2.561e-34	1	Large	0.000918	0.001942
7	4.140e-34	0.9984	Large	0.001429	0.002885
8	2.561e-34	1	Large	0.001944	0.004291
9	2.562e-34	1	Large	0.002051	0.005125
10	2.562e-34	1	Large	0.002648	0.006417
11	2.562e-34	1	Large	0.004188	0.008100
12	2.562e-34	1	Large	0.003967	0.009881
13	2.562e-34	1	Large	0.004379	0.010188
14	2.562e-34	1	Large	0.004862	0.011041
15	2.562e-34	1	Large	0.005565	0.012900
16	2.562e-34	1	Large	0.006656	0.014812

Table B.1: Effect size comparison descending order

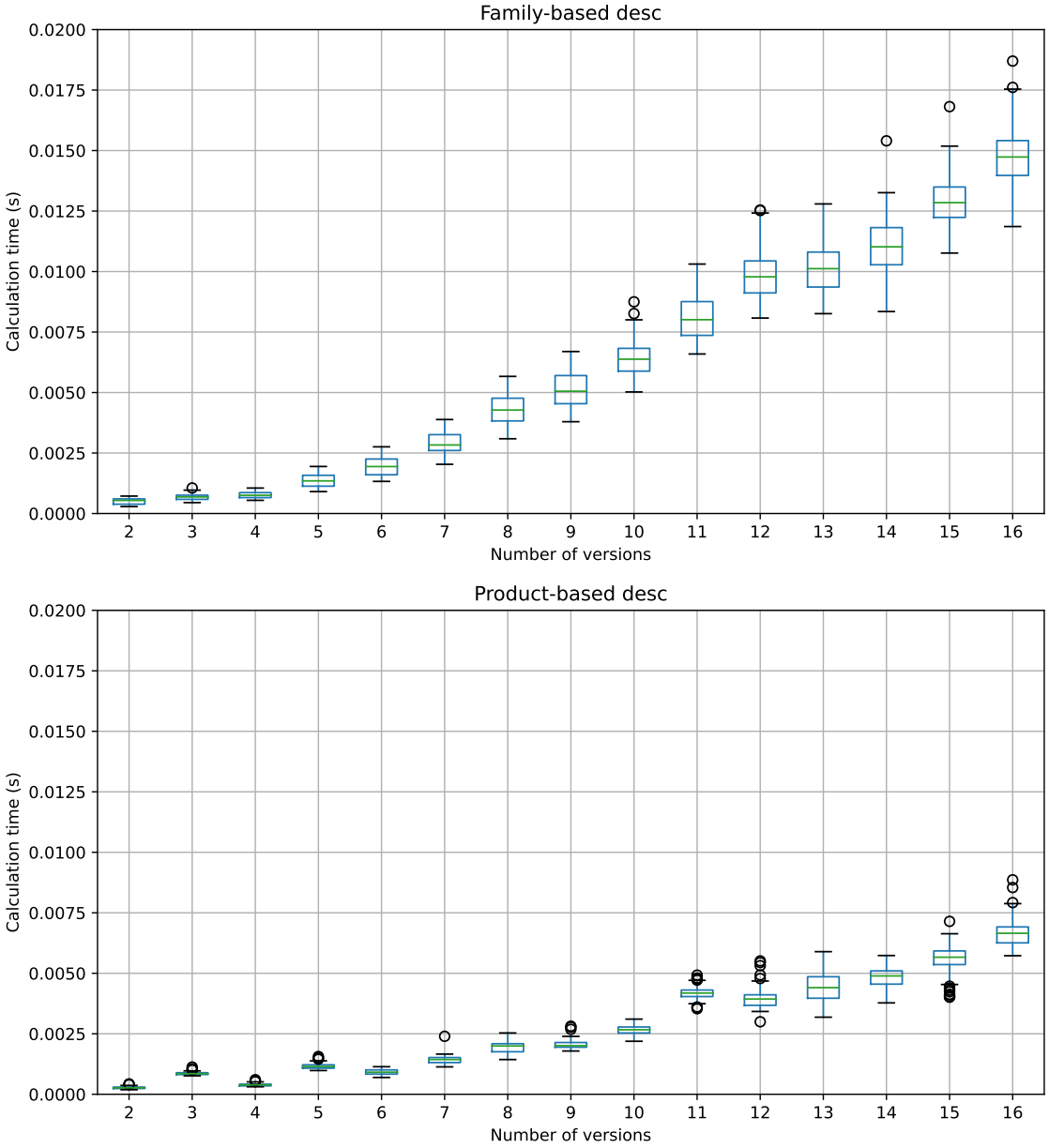


Figure B.1: Comparison family-based vs product-based desc

Appendix C

CADS vs CPDS comparison

	CPDS - time				CADS - time			
Nr	**Mean (s)**	**Std**	**Min (s)**	**Max (s)**	**Mean (s)**	**Std**	**Min (s)**	**Max (s)**
2	0.001166	0.000182	0.000999	0.001486	0.012065	0.003891	0.009760	0.020110
3	0.002343	0.000381	0.002087	0.003369	0.031932	0.004322	0.027148	0.039431
4	0.002453	0.000112	0.002327	0.002645	0.042078	0.020116	0.030047	0.080362
5	0.004452	0.000171	0.004158	0.004811	0.065908	0.002719	0.062587	0.070555
6	0.022725	0.000604	0.021845	0.023562	0.121627	0.003720	0.116302	0.129635
7	0.038360	0.002410	0.036055	0.044513	0.174496	0.006739	0.168172	0.188787
8	0.233280	0.027915	0.214656	0.309031	0.299723	0.024422	0.276643	0.341510
9	1.077473	0.093839	0.995781	1.204954	0.524601	0.068205	0.464019	0.666431
10	4.024490	0.086735	3.889179	4.131253	7.728839	0.102699	7.580769	7.881543
11	20.708756	1.946604	18.766176	24.942806	16.689287	2.058890	15.590138	22.262953
12	59.191610	13.062229	49.976879	93.228886	18.882142	1.045757	18.046849	21.565536
13	75.443015	6.847498	64.996472	86.828969	26.488269	1.836590	23.124847	29.952911
14	321.700199	21.390381	287.296967	366.103355	31.420006	1.803964	28.552680	33.941879
15	1132.775543	1.277020	1131.872554	1133.678532	49.832779	2.234924	46.478809	53.468031
16	1123.779180	23.180337	1093.579165	1147.945390	49.375182	3.168295	45.451823	55.017523

Table C.1: Time comparison per number of version

	CPDS - Avg. input				CADS - Avg. input			
Nr	**Mean**	**Std**	**Min**	**Max**	**Mean**	**Std**	**Min**	**Max**
2	4.000000	0.000000	4.000000	4.000000	4.000000	0.000000	4.000000	4.000000
3	5.000000	0.000000	5.000000	5.000000	4.000000	0.000000	4.000000	4.000000
4	5.000000	0.000000	5.000000	5.000000	3.500000	0.000000	3.500000	3.500000
5	6.000000	0.000000	6.000000	6.000000	3.800000	0.000000	3.800000	3.800000
6	10.000000	0.000000	10.000000	10.000000	7.000000	0.000000	7.000000	7.000000
7	10.000000	0.000000	10.000000	10.000000	6.285714	0.000000	6.285714	6.285714
8	14.000000	0.000000	14.000000	14.000000	7.825000	1.336195	6.750000	9.375000
9	17.000000	0.000000	17.000000	17.000000	8.922222	1.241763	7.333333	11.222222
10	21.000000	0.000000	21.000000	21.000000	12.430000	0.692901	11.600000	13.900000
11	23.000000	0.000000	23.000000	23.000000	12.081818	1.417748	10.363636	14.818182
12	27.000000	0.000000	27.000000	27.000000	14.808333	3.268124	10.666667	21.000000
13	27.000000	0.000000	27.000000	27.000000	13.984615	2.163835	11.692308	19.230769
14	27.000000	0.000000	27.000000	27.000000	12.542857	1.440506	10.714286	15.000000
15	29.000000	0.000000	29.000000	29.000000	13.953333	1.416325	12.400000	16.666667
16	29.000000	0.000000	29.000000	29.000000	14.768750	1.396004	12.812500	16.875000

Table C.2: Average number of inputs: CPDS vs CADS

	CPDS - Max. inputs				CADS - Max. inputs			
Nr	**Mean**	**Std**	**Min**	**Max**	**Mean**	**Std**	**Min**	**Max**
2	4.000000	0.000000	4	4	4.000000	0.000000	4	4
3	5.000000	0.000000	5	5	5.000000	0.000000	5	5
4	5.000000	0.000000	5	5	5.000000	0.000000	5	5
5	6.000000	0.000000	6	6	6.000000	0.000000	6	6
6	10.000000	0.000000	10	10	10.000000	0.000000	10	10
7	10.000000	0.000000	10	10	10.000000	0.000000	10	10
8	14.000000	0.000000	14	14	11.600000	2.065591	10	14
9	17.000000	0.000000	17	17	13.100000	2.330951	10	17
10	21.000000	0.000000	21	21	19.800000	1.398412	18	23
11	23.000000	0.000000	23	23	17.800000	2.250926	14	21
12	27.000000	0.000000	27	27	21.800000	4.894441	16	31
13	27.000000	0.000000	27	27	20.500000	3.341656	17	28
14	27.000000	0.000000	27	27	18.700000	2.496664	14	23
15	29.000000	0.000000	29	29	21.100000	3.071373	17	26
16	29.000000	0.000000	29	29	23.000000	2.000000	20	25

Table C.3: Maximal number of inputs: CPDS vs CADS

	CPDS - Avg. resets				CADS - Avg. resets			
Nr	**Mean**	**Std**	**Min**	**Max**	**Mean**	**Std**	**Min**	**Max**
2	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
3	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
4	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
5	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
6	1.000000	0.000000	1.000000	1.000000	0.833333	0.000000	0.833333	0.833333
7	1.000000	0.000000	1.000000	1.000000	0.714286	0.000000	0.714286	0.714286
8	2.000000	0.000000	2.000000	2.000000	1.125000	0.322749	0.875000	1.500000
9	3.000000	0.000000	3.000000	3.000000	1.488889	0.332096	1.111111	2.222222
10	4.000000	0.000000	4.000000	4.000000	2.300000	0.270801	2.100000	3.000000
11	5.000000	0.000000	5.000000	5.000000	2.245455	0.388186	1.727273	3.000000
12	5.000000	0.000000	5.000000	5.000000	2.775000	0.641961	1.916667	3.750000
13	5.000000	0.000000	5.000000	5.000000	2.646154	0.481342	2.076923	3.538462
14	5.000000	0.000000	5.000000	5.000000	2.407143	0.444225	1.857143	3.142857
15	6.000000	0.000000	6.000000	6.000000	3.120000	0.445332	2.600000	3.866667
16	6.000000	0.000000	6.000000	6.000000	3.406250	0.305121	2.812500	3.750000

Table C.4: Average number of resets: CPDS vs CADS

	CPDS - Max. resets				CADS - Max. resets			
Nr	**Mean**	**Std**	**Min**	**Max**	**Mean**	**Std**	**Min**	**Max**
2	0.000000	0.000000	0	0	0.000000	0.000000	0	0
3	0.000000	0.000000	0	0	0.000000	0.000000	0	0
4	0.000000	0.000000	0	0	0.000000	0.000000	0	0
5	0.000000	0.000000	0	0	0.000000	0.000000	0	0
6	1.000000	0.000000	1	1	1.000000	0.000000	1	1
7	1.000000	0.000000	1	1	1.000000	0.000000	1	1
8	2.000000	0.000000	2	2	1.400000	0.516398	1	2
9	3.000000	0.000000	3	3	2.100000	0.316228	2	3
10	4.000000	0.000000	4	4	3.700000	0.674949	3	5
11	5.000000	0.000000	5	5	3.300000	0.674949	2	4
12	5.000000	0.000000	5	5	4.300000	1.059350	3	6
13	5.000000	0.000000	5	5	3.900000	0.875595	3	5
14	5.000000	0.000000	5	5	3.600000	0.843274	2	5
15	6.000000	0.000000	6	6	4.800000	0.918937	4	6
16	6.000000	0.000000	6	6	5.100000	0.567646	4	6

Table C.5: Maximal number of resets: CPDS vs CADS

Appendix D

CADS vs family-based Online Machine Separation comparison

	Online Machine Separation - Time				CADS - Time			
Nr	**Mean (s)**	**Std**	**Min (s)**	**Max (s)**	**Mean (s)**	**Std**	**Min (s)**	**Max (s)**
2	0.001106	0.000139	0.000892	0.001484	0.010142	0.001331	0.009128	0.017396
3	0.001621	0.000182	0.001338	0.002329	0.028200	0.000945	0.027191	0.031829
4	0.001773	0.000163	0.001389	0.002216	0.031219	0.000635	0.030458	0.033511
5	0.003208	0.001589	0.002173	0.010752	0.071430	0.007830	0.065880	0.103454
6	0.003916	0.000451	0.002991	0.004666	0.125340	0.003864	0.120939	0.147238
7	0.004583	0.000567	0.003321	0.005688	0.180602	0.006946	0.174370	0.218928
8	0.006120	0.000522	0.004746	0.007511	0.275155	0.012157	0.264631	0.303203
9	0.007039	0.000582	0.005831	0.008241	0.499031	0.021431	0.479214	0.541760
10	0.008470	0.000524	0.007358	0.009470	7.916353	0.108207	7.745819	8.172039
11	0.009265	0.001090	0.007450	0.011464	16.023204	0.326786	15.746295	17.943678
12	0.011587	0.001168	0.009789	0.017140	18.604936	0.625742	17.851946	20.559991
13	0.012143	0.001098	0.010251	0.015273	21.873293	0.825066	21.047754	24.138849
14	0.013751	0.001485	0.011569	0.020530	27.989035	0.875418	26.980695	30.238187
15	0.014497	0.001543	0.011871	0.020980	44.526636	1.326488	42.919517	47.764718
16	0.015548	0.002141	0.012722	0.025188	45.954853	1.114407	44.168518	48.195820

Table D.1: Time comparison per number of version

	Online Machine Separation - Avg. inputs				CADS - Avg. inputs			
Nr	**Mean**	**Std**	**Min**	**Max**	**Mean**	**Std**	**Min**	**Max**
2	4.000000	0.000000	4.000000	4.000000	4.000000	0.000000	4.000000	4.000000
3	7.000000	0.000000	7.000000	7.000000	4.000000	0.000000	4.000000	4.000000
4	7.000000	0.000000	7.000000	7.000000	3.500000	0.000000	3.500000	3.500000
5	10.520000	1.110984	8.000000	11.000000	3.800000	0.000000	3.800000	3.800000
6	14.160000	1.360672	12.000000	15.000000	7.000000	0.000000	7.000000	7.000000
7	16.080000	1.988128	12.000000	18.000000	6.285714	0.000000	6.285714	6.285714
8	20.680000	1.731344	16.000000	22.000000	8.862500	1.293195	6.750000	10.375000
9	23.860000	1.807129	19.000000	25.000000	9.462222	1.332267	7.333333	11.222222
10	27.020000	1.671978	23.000000	29.000000	11.874000	1.245696	9.400000	14.800000
11	28.960000	3.057009	23.000000	34.000000	12.669091	1.469285	10.000000	16.181818
12	34.100000	2.704871	29.000000	40.000000	13.540000	2.029281	10.666667	20.666667
13	34.040000	2.913445	29.000000	40.000000	13.261538	1.970560	10.923077	20.384615
14	36.780000	2.541814	32.000000	43.000000	13.141429	1.813429	10.928571	20.785714
15	38.240000	3.007202	31.000000	45.000000	14.537333	1.697666	11.800000	19.400000
16	39.260000	2.746501	34.000000	45.000000	13.997500	1.395199	11.062500	16.500000

Table D.2: Average number of inputs: CADS vs Online Machine Separation

	Online Machine Separation - Max. inputs				CADS - Max. inputs			
Nr	**Mean**	**Std**	**Min**	**Max**	**Mean**	**Std**	**Min**	**Max**
2	4.000000	0.000000	4	4	4.000000	0.000000	4	4
3	7.000000	0.000000	7	7	5.000000	0.000000	5	5
4	7.000000	0.000000	7	7	5.000000	0.000000	5	5
5	10.520000	1.110984	8	11	6.000000	0.000000	6	6
6	14.160000	1.360672	12	15	10.000000	0.000000	10	10
7	16.080000	1.988128	12	18	10.000000	0.000000	10	10
8	20.680000	1.731344	16	22	13.120000	1.902093	10	15
9	23.860000	1.807129	19	25	13.920000	2.397618	10	17
10	27.020000	1.671978	23	29	19.020000	2.254157	14	24
11	28.960000	3.057009	23	34	18.720000	2.634465	14	25
12	34.100000	2.704871	29	40	19.460000	3.078629	14	29
13	34.040000	2.913445	29	40	19.520000	3.124884	16	29
14	36.780000	2.541814	32	43	19.440000	3.084921	16	31
15	38.240000	3.007202	31	45	21.580000	2.595837	18	28
16	39.260000	2.746501	34	45	21.720000	2.835921	16	28

Table D.3: Maximal number of inputs: CADS vs Online Machine Separation

Nr	Online Machine Separation - Avg. resets				CADS - Avg. resets			
	Mean	**Std**	**Min**	**Max**	**Mean**	**Std**	**Min**	**Max**
2	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
3	1.000000	0.000000	1.000000	1.000000	0.000000	0.000000	0.000000	0.000000
4	1.000000	0.000000	1.000000	1.000000	0.000000	0.000000	0.000000	0.000000
5	1.840000	0.370328	1.000000	2.000000	0.000000	0.000000	0.000000	0.000000
6	2.720000	0.453557	2.000000	3.000000	0.833333	0.000000	0.833333	0.833333
7	3.360000	0.662709	2.000000	4.000000	0.714286	0.000000	0.714286	0.714286
8	4.560000	0.577115	3.000000	5.000000	1.337500	0.276930	0.875000	1.500000
9	5.620000	0.602376	4.000000	6.000000	1.604444	0.376401	1.111111	2.222222
10	6.340000	0.557326	5.000000	7.000000	2.246000	0.356405	1.600000	3.000000
11	6.760000	0.796933	5.000000	8.000000	2.365455	0.406346	1.727273	3.181818
12	7.580000	0.784805	6.000000	9.000000	2.498333	0.435294	1.833333	3.583333
13	7.600000	0.832993	6.000000	9.000000	2.484615	0.410951	1.923077	3.538462
14	8.460000	0.705951	7.000000	10.000000	2.511429	0.417585	1.928571	3.714286
15	9.320000	0.935469	7.000000	11.000000	3.265333	0.392328	2.200000	3.866667
16	9.540000	0.787919	8.000000	11.000000	3.228750	0.395206	2.062500	3.750000

Table D.4: Average number of resets: CADS vs Online Machine Separation

Nr	Online Machine Separation - Max. resets				CADS - Max. resets			
	Mean	**Std**	**Min**	**Max**	**Mean**	**Std**	**Min**	**Max**
2	0.000000	0.000000	0	0	0.000000	0.000000	0	0
3	1.000000	0.000000	1	1	0.000000	0.000000	0	0
4	1.000000	0.000000	1	1	0.000000	0.000000	0	0
5	1.840000	0.370328	1	2	0.000000	0.000000	0	0
6	2.720000	0.453557	2	3	1.000000	0.000000	1	1
7	3.360000	0.662709	2	4	1.000000	0.000000	1	1
8	4.560000	0.577115	3	5	1.740000	0.443087	1	2
9	5.620000	0.602376	4	6	2.220000	0.418452	2	3
10	6.340000	0.557326	5	7	3.600000	0.699854	2	5
11	6.760000	0.796933	5	8	3.480000	0.838852	2	5
12	7.580000	0.784805	6	9	3.580000	0.730949	2	5
13	7.600000	0.832993	6	9	3.760000	0.743955	3	5
14	8.460000	0.705951	7	10	4.020000	0.769044	3	6
15	9.320000	0.935469	7	11	4.740000	0.664247	4	6
16	9.540000	0.787919	8	11	4.900000	0.839096	3	7

Table D.5: Maximal number of resets: CADS vs Online Machine Separation