AIM: Automated Input Set Minimization for Metamorphic Security Testing

NAZANIN BAYATI CHALESHTARI, University of Ottawa, Canada
YOANN MARQUER, SnT Centre, University of Luxembourg, Luxembourg
FABRIZIO PASTORE, SnT Centre, University of Luxembourg, Luxembourg
LIONEL C. BRIAND, University of Ottawa, Canada and Lero SFI Centre for Software Research and University of Limerick, Ireland

Although the security testing of Web systems can be automated by generating crafted inputs, solutions to automate the test oracle, i.e., distinguishing correct from incorrect outputs, remain preliminary. Specifically, previous work has demonstrated the potential of metamorphic testing; indeed, security failures can be determined by metamorphic relations that turn valid inputs into malicious inputs. However, without further guidance, metamorphic relations are typically executed on a large set of inputs, which is time-consuming and thus makes metamorphic testing impractical.

We propose AIM, an approach that automatically selects inputs to reduce testing costs while preserving vulnerability detection capabilities. AIM includes a clustering-based black box approach, to identify similar inputs based on their security properties. It also relies on a novel genetic algorithm able to efficiently select diverse inputs while minimizing their total cost. Further, it contains a problem-reduction component to reduce the search space and speed up the minimization process. We evaluated the effectiveness of AIM on two well-known Web systems, Jenkins and Joomla, with documented vulnerabilities. We compared AIM's results with four baselines. Overall, AIM reduced metamorphic testing time by 84% for Jenkins and 82% for Joomla, while preserving vulnerability detection. Furthermore, AIM outperformed all the considered baselines regarding vulnerability coverage.

CCS Concepts: • **Software and its engineering** → **Software verification and validation**; • **Security and privacy** → *Web application security*; • **Computing methodologies** → *Genetic algorithms*; • **Information systems** → Clustering.

Additional Key Words and Phrases: Automated Security Testing, Metamorphic Testing, Input set Minimization

ACM Reference Format:
Nazanin Bayati Chaleshtari, Yoann Marquer, Fabrizio Pastore, and Lionel C. Briand. 2024. AIM: Automated Input Set Minimization for Metamorphic Security Testing. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (February 2024), 59 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Authors' addresses: Nazanin Bayati Chaleshtari, University of Ottawa, School of Electrical and Computer Engineering, Canada, n.bayati@uottawa.ca; Yoann Marquer, SnT Centre, University of Luxembourg, Luxembourg, yoann.marquer@uni.lu; Fabrizio Pastore, SnT Centre, University of Luxembourg, Luxembourg, fabrizio.pastore@uni.lu; Lionel C. Briand, University of Ottawa, School of Electrical and Computer Engineering, Canada, lbriand@uottawa.ca and Lero SFI Centre for Software Research and University of Limerick, Ireland, lionel.briand@lero.ie.

1049-331X/2024/2-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACKNOWLEDGMENTS

This work is supported by the H2020 COSMOS European project, grant agreement No. 957254, the Science Foundation Ireland grant 13/RC/2094-2, and NSERC of Canada under the Discovery and CRC programs.

1 INTRODUCTION

Web systems, from social media platforms to e-commerce and banking systems, are a backbone of our society: they manage data that is at the heart of our social and business activities (e.g., public pictures, bank transactions), and, as such, should be protected. To verify that Web systems are secure, engineers perform security testing, which consists of verifying that the software adheres to its security properties (e.g., confidentiality, availability, and integrity). Such testing is typically performed by simulating malicious users interacting with the system under test [36, 37].

At a high-level, security testing does not differ from other software testing activities: it consists of providing inputs to the software under test and verifying that the software outputs are correct, based on specifications. For such a verification, a *test oracle* [27] is required, i.e., a mechanism for determining whether a test case has passed or failed. When test cases are manually derived, test oracles are defined by engineers and they generally consist of expressions comparing an observed output with the expected output, determined from software specifications. In security testing, when a software output differs from the expected one, then a software vulnerability (i.e., a fault affecting the security properties of the software) has been discovered.

Automatically deriving test oracles for the software under test (SUT) is called the *oracle problem* [8], which entails distinguishing correct from incorrect outputs for all potential inputs. Except for the verification of basic reliability properties—ensuring that the software provides a timely output and does not crash—the problem is not tractable without additional executable specifications (e.g., method post-conditions or detailed system models), which, unfortunately, are often unavailable. Further, since software vulnerabilities tend to be subtle, it is necessary to exercise each software interface with a large number of inputs (e.g., providing all the possible code injection strings to a Web form). When a large number of test inputs are needed, even in the presence of automated means to generate them (e.g., catalogs of code injection strings), testing becomes impractical if we lack solutions to automatically derive test oracles.

Metamorphic testing (MT) was proposed to alleviate the oracle problem [57] by testing not the input-output behavior of the system, but by comparing the outputs of multiple test executions [11, 57]. It relies on metamorphic relations (MRs), which are specifications expressing relations between outputs under certain input conditions, which are then used to derive a follow-up input from a source input to satisfy such relations. Such an approach has shown to be useful for security testing, also referred to as metamorphic security testing (MST) [11, 38]. MST consists in relying on MRs to modify source inputs to obtain follow-up inputs that mimic attacks and verify that known output properties captured by these MRs hold (e.g., if the follow-up input differs from the source input in some way, then the output shall be different). For instance, one may verify if URLs can be accessed by users who can't reach them through their user interface, thus enabling the detection of authorization vulnerabilities.

MST has been successfully applied to testing Web interfaces [11, 38] in an approach called MST-wi [11]; in such context, source inputs are sequences of interaction with a Web system and can be easily derived using a Web crawler. For example, a source input may consist of two actions: performing a login, and requesting a specific URL appearing in the returned Web page. MST-wi integrates a catalog of 76 MRs enabling the identification of 101 vulnerability types.

The performance and scalability of metamorphic testing naturally depends on the number of source inputs to be processed. In the case of MST-wi, we demonstrated that scalability can be achieved through parallelism; however, such solution may not fit all development contexts (e.g., not all the companies have an infrastructure enabling parallel execution of the software under test and its test cases). Further, even when parallelization is possible, a reduction of the test execution time may provide tangible benefits, including earlier vulnerability detection. In general, what is required is an approach to minimize the number of source inputs to be used during testing.

In this work, we address the problem of minimizing source inputs used by MRs to make MST scalable, with a focus on Web systems, though many aspects are reusable to other domains. We propose the Automated Input Minimizer (AIM) approach, which aims at minimizing a set of source inputs (hereafter, the *initial input set*), while preserving the capability of MRs to detect security vulnerabilities. This work includes the following contributions:

- We propose AIM, an approach to minimize input sets for metamorphic testing while retaining inputs able to exercise vulnerabilities. Note that many steps of AIM are not specific to Web systems while others would need to be tailored to other domains (e.g., desktop applications, embedded systems). This approach includes the following novel components:
  - An extension of the MST-wi framework to retrieve output data and extract cost information about MRs without executing them.
  - A black-box approach leveraging clustering algorithms to partition the initial input set based on security-related characteristics.
  - MOCCO (Many-Objective Coverage and Cost Optimizer), a novel genetic algorithm which is able to efficiently select diverse inputs while minimizing their total cost.
  - IMPRO (Inputset Minimization Problem Reduction Operator), an approach to reduce the search space to its minimal extent, then divide it in smaller independent parts.
- We present a prototype framework for AIM [14], integrating the above components and automating the process of input set minimization for Web systems.
- We report on an extensive empirical evaluation aimed at assessing the effectiveness of AIM in terms of vulnerability detection and performance, considering 18 different AIM configurations and 4 common baselines for security testing, on the Jenkins and Joomla systems, which are the most used Web-based frameworks for development automation and context management.
- We also provide a proof of the correctness of the AIM approach (Appendices A and B).

This paper is structured as follows. We introduce background information necessary to state our problem and detail our approach (Section 2). We define the problem of minimizing the initial input set while retaining inputs capable of detecting distinct software vulnerabilities (Section 3). We present an overview of AIM (Section 4) and then detail our core technical solutions (Sections 5 to 9). We report on a large-scale empirical evaluation of AIM (Section 10) and address the threats to the validity of the results (Section 11). We discuss and contrast related work (Section 12) and draw conclusions (Section 13).

2 BACKGROUND

In this section, we present the concepts required to define our approach. We first provide a background on Metamorphic Testing (MT, § 2.1), then we briefly describe MST-wi, our previous work on the application of MT to security (§ 2.2). Next, we introduce optimization problems (§ 2.3). Finally, we briefly describe three clustering algorithms: K-means (§ 2.4.1), DBSCAN (§ 2.4.2), and HDBSCAN (§ 2.4.3).

2.1 Metamorphic Testing

In contrast to common testing practice, which compares for each input of the system the actual output against the expected output, MT examines the relationships between outputs obtained from multiple test executions.

MT is based on Metamorphic Relations (MRs), which are necessary properties of the system under test in relation to multiple inputs and their expected outputs [16]. The test result, either pass or failure, is determined by validating the outputs of various executions against the MR.

Formally, let S be the system under test. In the context of MT, inputs in the domain of S are called *source inputs*. Moreover, we call *source output* and we denote $S(x)$ the output obtained from a source input x. An MR is the combination of:

- A *transformation function* θ, taking values in source inputs and generating new inputs called *follow-up inputs*. For each source input x, we call *follow-up output* the output $S(\theta(x))$ of the follow-up input $\theta(x)$.
- An *output relation* R between source outputs and follow-up outputs. For each source input x, if $R(S(x), S(\theta(x)))$ holds, then the test passes, otherwise the test fails.

We now provide an example to further clarify the concepts presented above.

Example 1. Consider a system implementing the cosine function. It might not be feasible to verify the $\cos(x)$ results for all possible values of x, except for special values of x, e.g. $cos(0) = 1$ or $cos(\frac{\pi}{2}) = 0$. However, the cosine function satisfies that, for each input x, $\cos(\pi - x) = -\cos(x)$. Based on this property, we can define an MR, where the source inputs are the possible angle values of x, the follow-up inputs are $y = \pi - x$, and the expected relation between source and follow-up outputs is $\cos(y) = -\cos(x)$. The system is executed twice, respectively with an angle x and an angle $y = \pi - x$. The outputs of both executions are then validated against the output relation. If this relation is violated, then the system is faulty.

2.2 Metamorphic Security Testing

In our previous work, we automated MT in the security domain by introducing a tool named MST-wi [11]. MST-wi enables software engineers to define MRs that capture the security properties of Web systems. MST-wi includes a data collection framework that crawls the Web system under test to automatically derive source inputs. Each source input is a sequence of interactions of the legitimate user with the Web system. Also, MST-wi includes a Domain Specific Language (DSL) to support writing MRs for security testing. Moreover, MST-wi provides a testing framework that automatically performs security testing based on the defined MRs and the input data.

In MST, follow-up inputs are generated by modifying source inputs, simulating actions an attacker might take to identify vulnerabilities in the system. These modifications can be done using 55 Web-specific functions enabling engineers to define complex security properties, e.g., `cannotReachThroughGUI`, `isSupervisorOf`, and `isError`. MRs capture security properties that hold when the system behaves in a safe way. If an MR, for any given input, gets violated, then MST-wi detected a vulnerability in the system. MST-wi includes a catalogue of 76 MRs, inspired by OWASP guidelines [53] and vulnerability descriptions in the CWE database [43], capturing a large variety of security properties for Web systems.

Example 2. We describe in Figure 1 an MR written for CWE_286, which concerns unintended access rights [41]. This MR tests if a user navigating the GUI cannot access a URL, then the same URL should not be available to this user when she directly requests it from the server. A source input is a sequence of actions. The for loop iterates over all the actions of an input (Line 3). This includes an action y that the user cannot access while navigating the GUI. The function `User()`

```
1  MR CWE_286_OTG_AUTHZ_002c {
2    {
3      for(var y = Input(1).actions().size()-1; ( y > 0 ); y--){
4          IMPLIES(
5              (!isSupervisorOf(User(), Input(1).actions().get(y).user)) &&
6              afterLogin(Input(1).actions().get(y)) &&
7              cannotReachThroughGUI(User(),
8                     Input(1).actions().get(y).getUrl()) &&
9              CREATE(Input(2), Input(LoginAction(User()),
10                    Input(1).actions().get(y)))
11             ,
12          OR(
13             isError(Output(Input(1), y)),
14             different(Output(Input(1), y),Output(Input(2), 1)))
15     ); //end-IMPLIES
16    } //end-for
17   }
18  } //end-MR
19 } //end-package
```

Fig. 1. MR CWE_286, testing for incorrect user management

returns a (randomly selected) user having an account on the system, we call it follow-up user. The MR first checks whether the follow-up user is not a supervisor of the user performing the y-th action (Line 5), because a supervisor might have direct access to the action's URL. Then the MR checks that the y-th action is performed after a login (Line 6), to ensure this action requires authentication. Also, the MR checks that the follow-up user cannot retrieve the URL of the action through the GUI (Line 7), based on the data collected by the crawler. The MR defines the follow-up input as a direct request for the action y, after performing the login as the follow-up user (Line 9). If the system is secure, it should display different outputs for the source and follow-up inputs to the user, or it should show an error message for the follow-up input. Therefore, the MR checks if the y-th action from the source input leads to an error page (Line 13) or if the generated outputs are different (Line 14).

2.3 Many Objective Optimization

Engineers are often faced with problems requiring to fulfill multiple objectives at the same time, called *multi-objective problems*. For instance, when generating a new test suite, one may face tension between the number of test cases and the number of covered branches in the source code. Multi-objective problems with at least (three or) four objectives are informally known as *many-objective problems* [34]. In both kind of problems, one need a solution which is a good trade-off between the objectives. Hence, we first introduce the Pareto front of a decision space (§ 2.3.1). Then, we describe genetic algorithms able to solve many-objective problems (§ 2.3.2).

2.3.1 Pareto Front. Multi- and many-objective problems can be stated as minimizing several objective functions while taking values in a given decision space. The goal of multi-objective optimization is to approximate the Pareto Front in the objective space [34].

Formally, if D is the decision space and $f_1(.), \dots, f_n(.)$ are n objective functions defined on D, then the *fitness vector* of a decision vector $x \in D$ is $[f_1(x), \dots, f_n(x)]$, hereafter denoted $F(x)$. Moreover, a decision vector x_1 *Pareto-dominates* a decision vector x_2 (hereafter denoted $x_1 \succ x_2$) if 1) for each $1 \leq i \leq n$, we have $f_i(x_1) \leq f_i(x_2)$, and 2) there exists $1 \leq i \leq n$ such that $f_i(x_1) < f_i(x_2)$. If there exists no decision vector x_1 such that $x_1 \succ x_2$, we say that the decision vector x_2 is *non-dominated*. The *Pareto front* of D is the set $\{F(x_2) \mid x_2 \in D \text{ and } \forall x_1 \in D : x_1 \nsucc x_2\}$ of the fitness vectors of the non-dominated decision vectors. Finally, a *multi/many-objective problem* consists in:

$$\underset{x \in D}{\text{minimize}}\, F(x) = [f_1(x), \dots, f_n(x)]$$

where the minimize notation means that we want to find or at least approximate the non-dominated decision vectors, hence the ones having a fitness vector in the Pareto front [34].

2.3.2 Solving Many-Objective Problems. Multi-objective algorithms like NSGA-II [19] or SPEA2 [31, 70] are not effective in solving many-objective problems [18, 54] because of the following challenges:

(1) The proportion of non-dominated solutions becomes exponentially large with an increased number of objectives. This reduces the chances of the search being stuck at a local optimum and may lead to a better convergence rate [34], but also slows down the search process considerably [18].
(2) With an increased number of objectives, diversity operators (e.g., based on crowding distance or clustering) become computationally expensive [18].
(3) If only a handful of solutions are to be found in a large-dimensional space, solutions are likely to be widely distant from each other. Hence, two distant parent solutions are likely to produce offspring solutions that are distant from them. In this situation, recombination operations may be inefficient and require crossover restriction or other schemes [18].

To tackle these challenges, several many-objective algorithms have been successfully applied within the software engineering community, like NSGA-III [18, 29] and MOSA [54].

NSGA-III [18, 29] is based on NSGA-II [19] and addresses these challenges by assuming a set of supplied or predefined reference points. Diversity (challenge 2) is ensured by starting the search in parallel from each of the reference points, assuming that largely spread starting points would lead to exploring all relevant parts of the Pareto front. For each parallel search, parents share the same starting point, so they are assumed to be close enough so that recombination operations (challenge 3) are more meaningful. Finally, instead of considering all solutions in the Pareto front, NSGA-III focuses on individuals which are the closest to the largest number of reference points. That way, NSGA-III considers only a small proportion of the Pareto front (addressing challenge 1).

Another many-objective algorithm, MOSA [54], does not aim to identify a single individual achieving a tradeoff between objectives but a set of individuals, each satisfying one of the objectives. Such characteristic makes MOSA adequate for many software testing problems where it is sufficient to identify one test case (i.e., an individual) for each test objective (e.g., covering a specific branch or violating a safety requirement). To deal with challenge 1, MOSA relies on a preference criterion amongst individuals in the Pareto front, by focusing on 1) extreme individuals (i.e., test cases having one or more objective scores equal to zero), and 2) in case of tie, the shortest test cases. These best extreme individuals are stored in an archive during the search, and the archive obtained at the last generation is the final solution. Challenges 2 and 3 are addressed by focusing the search, on each generation, on the objectives not yet covered by individuals in the archive.

2.4 Clustering

During the clustering steps in Section 6, we rely on three well-known clustering algorithms: K-means (§ 2.4.1), DBSCAN (§ 2.4.2), and HDBSCAN (§ 2.4.3).

2.4.1 K-means. K-means is a clustering algorithm which takes as input a set of data points and an integer K. K-means aims to assign data points to K clusters by maximizing the similarity between individual data points within each cluster and the center of the cluster, called centroid. The centroids are randomly initialized, then iteratively refined until a fixpoint is reached [4].

2.4.2 DBSCAN. DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is an algorithm that defines clusters using local density estimation. This algorithm takes as input a dataset and two configuration parameters: the distance threshold ϵ and the minimum number of neighbours n.

The distance threshold ϵ is used to determine the ϵ-neighbourhood of each data point, i.e., the set of data points that are at most ϵ distant from it. There are three different types of data points in DBSCAN, based on the number of neighbours in the ϵ-neighbourhood of a data point:

Core point. If a data point has a number of neighbours above n, it is then considered a core point.

Border point. If a data point has a number of neighbours below n, but has a core point in its neighborhood, it is then considered a border point.

Noise. Any data point which is neither a core point nor a border point is considered noise.

A cluster consists of the set of core points and border points that can be reached through their ϵ-neighbourhoods [21]. DBSCAN uses a single global ϵ value to determine the clusters. But, if the clusters have varying densities, this could lead to suboptimal partitioning of the data. HDBSCAN addresses this problem and we describe next.

2.4.3 HDBSCAN. HDBSCAN (Hierarchical Density-Based Spatial Clustering of Applications with Noise) is an extension of DBSCAN (§ 2.4.2). As opposed to DBSCAN, HDBSCAN relies on different distance thresholds ϵ for each cluster, thus obtaining clusters of varying densities.

HDBSCAN first builds a hierarchy of clusters, based on various ϵ values selected in decreasing order. Then, based on such a hierarchy, HDBSCAN selects as final clusters the most persistent ones, where cluster persistence represents how long a cluster remains the same without splitting when decreasing the value of ϵ. In HDBSCAN, one has only to specify one parameter, which is the minimum number of individuals required to form a cluster, denoted by n [39]. Clusters with less than n individuals are considered noise and ignored.

3 PROBLEM DEFINITION

We aim to minimize the set of source inputs to be used when applying MST to a Web system, given a set of MRs, such that we do not reduce its vulnerability detection capability. To ensure that a minimized input set can exercise the same vulnerabilities as the original one, intuitively, we should ensure that they exercise the same *input blocks*. In software testing, after identifying an important characteristic to consider for the inputs, one can partition the input space in blocks, i.e., pairwise disjoint sets of inputs, such that inputs in the same block exercise the SUT in a similar way [1].

As the manual identification of relevant input blocks for a large system is extremely costly, we rely on clustering for that purpose (Section 6). Moreover, since we consider several characteristics, based on input parameters and the outputs of the system, we obtain several partitions. Thus, one input can exercise several input blocks. In the rest of the paper, we rely on the notion of *input coverage*, indicating the input blocks being exercised by an input. For the problem we consider, we first assume we know, for each input in the initial input set, the cost and coverage of this input. Then, we state and motivate our goals (§ 3.1): identify a subset of the initial input set that minimizes total cost while maintaining the same coverage as the initial input set. We explain why we consider each input block to be covered as an individual objective (§ 3.2), hence facing a many-objective optimization problem. Then, we define the objective functions we consider for each individual objective (§ 3.3). Finally, we describe the solutions to our many-objective problem (§ 3.4).

3.1 Assumptions and Goals

We assume we know, for each input in the initial input set, 1) its cost and 2) its coverage.

1) Because we want to make MST scalable, the cost $cost(in)$ of an input in corresponds to the execution time required to verify if the considered MRs are satisfied with this input. Because we aim to reduce this execution time without having to execute the MRs, as it would defeat the purpose of input set minimization, we must determine a cost metric which is a good surrogate of execution time. We tackle this problem in § 5.1 and assume for now that the cost of an input is known. The total cost of an input set I is $cost(I) \stackrel{\text{def}}{=} \sum_{in \in I} cost(in)$.

2) To minimize the cost of metamorphic testing, we remove unnecessary inputs from the initial input set, but we want to preserve all the inputs able to exercise distinct vulnerabilities. Hence, we consider, for each initial input in, its coverage $Coverage(in)$. In our study, $Coverage(in)$ is the set of input blocks in exercises, and we determine these input blocks in Section 6 using *double-clustering*. In short, each input is a sequence of actions used to communicate with the Web system and each action leads to a different Web page. Since we focus on security vulnerabilities in Web systems we consider as input characteristics 1) the system outputs since they characterize system states, and 2) action parameters (e.g., the URL, values belonging to a submitted form, or the method of sending a request to the server) since vulnerabilities might be detected through specific combinations of parameter values. We first cluster the system outputs (i.e., textual content extracted from Web pages) to obtain output classes. Then, for each output class, we use action parameters to cluster the actions producing outputs in the class, obtaining action subclasses. Finally, we define the coverage of each input (i.e., each sequence of actions) as the subclasses exercised by the actions in the sequence. For now, we assume that the coverage of an input is known. The total coverage of an input set I is $Coverage(I) \stackrel{\text{def}}{=} \cup_{in \in I} Coverage(in)$.

We can now state our goals. We want to obtain a subset $I_{final} \subseteq I_{init}$ of the initial inputs such that 1) I_{final} does not reduce total input coverage, i.e., $Coverage(I_{final}) = Coverage(I_{init})$ and 2) I_{final} has minimal cost, i.e., $cost(I_{final}) = \min\{cost(I) \mid I \subseteq I_{init} \land Coverage(I) = Coverage(I_{init})\}$. Note that a solution I_{final} may not be necessarily unique.

3.2 A Many-Objective Problem

To minimize the initial input set, we focus on the *selection* of inputs that exercise the same input blocks in the initial input set. A potential solution to our problem is an input set $I \subseteq I_{init}$. Obtaining a solution I able to reach full input coverage is straightforward since, for each block bl, one can simply select an input in $Inputs(bl) \stackrel{\text{def}}{=} \{in \in I_{init} \mid bl \in Coverage(in)\}$. The hard part of the problem we face is to determine a combination of inputs able to reach full input coverage at a minimal cost. Hence, we have to consider an input set as a whole and not focus on individual inputs.

This is similar to the *whole suite* approach [23] targeting white-box testing. They use as objective the total number of covered branches. But, in our context, counting the number of uncovered blocks would consider as equivalent input sets that miss the same number of blocks, without taking into account that it may be easier to cover some blocks than others (e.g., some blocks may be covered by many inputs, but some only by a few) or that a block may be covered by inputs with different costs. Thus, to obtain a combination of inputs that minimize cost while preserving input coverage, we have to investigate how input sets cover each input block.

Hence, we are interested in covering each input block as an individual objective, in a way similar to the coverage of each code branch for white-box testing [54]. Because the total number of blocks to be covered is typically large (≥ 4), we deal with a *many-objective problem* [34]. This can be an advantage, because a many-objective reformulation of complex problems can reduce the probability of being trapped in local optima and may lead to a better convergence rate [54]. But this raises several challenges (§ 2.3.2) that we tackle while presenting our search algorithm (Section 8).

3.3 Objective Functions

To provide effective guidance to a search algorithm, we need to quantify when an input set is closer to the objective of covering a particular block *bl* than another input set. In other words, if I_1 and I_2 are two input sets which do not cover *bl* but have the same cost, we need to determine which one is more desirable to achieve the goals introduced in § 3.1 by defining appropriate objective functions.

In this example, I_1 and I_2 would only differ with respect to their particular combination of inputs and what could happen when extending one or the other by an input $in \in Inputs(bl)$. Adding this input would not only cover *bl*, but would also likely cover other blocks, that would then be covered by several inputs. To keep track of how a given block *bl* is covered by inputs from a given input set I, we introduce the concept of *superposition* as $superpos(bl, I) \overset{\text{def}}{=} \text{card}(Inputs(bl) \cap I)$. For instance, if $superpos(bl, I) = 1$, then there is only one input in I covering *bl*. In that case, this input is necessary to maintain the coverage of I. More generally, we quantify how much an input is necessary to ensure the coverage of an input set with the *redundancy* metric:

$$redundancy(in, I) \overset{\text{def}}{=} \min\{superpos(bl, I) \mid bl \in Coverage(in)\} - 1$$

The -1 is used to normalize the redundancy metric so that its range starts at 0. If $redundancy(in, I) = 0$, we say that *in* is *necessary* in I, otherwise we say that *in* is *redundant* in I. In the following, we denote $Redundant(I) \overset{\text{def}}{=} \{in \in I \mid redundancy(in, I) > 0\}$ the set of the redundant inputs in I.

To focus on the least costly input sets during the search (Section 8), we quantify the gain obtained by removing redundant inputs. If I contains a redundant input *in*, then we call *removal step* a transition from I to $I \setminus \{in\}$. Otherwise, we say that I is already *reduced*. Unfortunately, given two redundant inputs in_1 and in_2, removing in_1 may render in_2 necessary. Hence, when considering potential removal steps (e.g., removing either in_1 or in_2), one has to consider the order of these steps. We represent a *valid order* of removal steps by a list of inputs $[in_1, \ldots, in_n]$ to be removed from I such that, for each $0 \leq i < n$, in_{i+1} is redundant in $I \setminus \{in_1, \ldots, in_i\}$. We denote $ValidOrders(I)$ the set of valid orders of removal steps in I. Removing redundant inputs $in_1, \ldots, in_n$ leads to a reduction of cost $cost(in_1) + \cdots + cost(in_n)$. For each input set I, we consider the maximal *gain* from valid orders of removal steps:

$$gain(I) \overset{\text{def}}{=} \max\{\sum_{1 \leq i \leq n} cost(in_i) \mid [in_1, \ldots, in_n] \in ValidOrders(I)\}$$

To reduce the cost of computing this gain, we prove in the appendix (Theorem 1) that, to determine which orders of removal steps are valid, we can remove inputs in any arbitrary order, without having to resort to backtracking to previous inputs. Moreover, in our approach, we need to compute the gain only in situations when the number of redundant inputs is small. Indeed, during the search (Section 8), we do not consider input sets in general but input sets that are already reduced, to or from which we then add or remove only a few inputs. More precisely, the gain is only used to obtain the best removal steps while initializing the populations (§ 8.3) or mutating the offspring (§ 8.5), and to compute the objective functions (§ 8.6). Therefore, in our approach, exhaustively computing the gain is tractable.

To define our objective functions, let us consider an input set $I \subseteq I_{init}$ which is already reduced but does not cover a given input block *bl*. As described in § 3.2, any input $in_1 \in Inputs(bl)$ could be added to I in order to cover *bl*. In that case, inputs in I that cover a block in common with in_1 could become redundant, which would lead to a gain $gain(I \cup \{in_1\})$ after the corresponding removal steps. But adding in_1 to I would also result in additional cost, hence warranting we consider the benefit-cost balance $gain(I \cup \{in_1\}) - cost(in_1)$ to evaluate how efficiently bl is covered by in_1. More generaly, we define the *potential* of bl to be covered in I as the maximum benefit-cost balance obtained that way. But, as an input in_1 may be necessary to cover bl while leading to no or not enough removal steps, $gain(I \cup \{in_1\}) - cost(in_1)$ may be negative. Since we want to normalize the potential using a function that assumes its value to be ≥ 0, we shift all the benefit-cost balances for a given objective bl by adding a dedicated term. As the potential is a maximum, the worst case is when $gain(I \cup \{in_1\}) = 0$ and $cost(in_1)$ is the minimal cost amongst the inputs able to cover bl. Hence, we obtain the following definition for the potential of covering bl in I:

$$\begin{aligned} potential(I, bl) \stackrel{\text{def}}{=} & \max\{gain(I \cup \{in_1\}) - cost(in_1) \mid in_1 \in Inputs(bl)\} \\ & + \min\{cost(in_2) \mid in_2 \in Inputs(bl)\} \end{aligned}$$

This metric is more meaningful when the considered input set is already reduced since, in that case, the gain would come only from reductions obtained after adding inputs able to cover the gap. If an input set is not already reduced, then unrelated inputs contribute to the potential of the input set, hence blurring the information regarding the block to be covered. This is why during the search (Section 8) we consider as candidates only reduced input sets and, consequently, we reduce each input set after an input is added.

We consider again an input block bl and two reduced input sets I_1 and I_2 , both of them not covering bl but having the same cost. If $potential(I_1, bl) > potential(I_2, bl)$ then, due to its combination of inputs, I_1 can be extended to cover bl in a way that (after removal steps) is less costly than it would be for I_2. Thus, we consider I_1 to be a more desirable way to achieve the coverage of bl than I_2. In other words, we use the potential to define the objective function associated with objective bl.

To normalize our metrics, we rely on the normalization function $\omega(x) \stackrel{\text{def}}{=} \frac{x}{x+1}$, which is used to reduce a range of values from $[0, \infty)$ to $[0, 1)$ while preserving the ordering. When used during a search, it is less prone to precision errors and more likely to drive faster convergence towards an adequate solution than alternatives [2]. We use it to normalize the cost $\omega(cost(I))$ between 0 and 1 and the smaller is the normalized cost, the better a solution is. For coverage objectives, since a high potential is more desirable, we use its complement $\frac{1}{x+1} = 1 - \omega(x)$ to reverse the order, so that the more potential an input set has, the lower its coverage objective function is. We denote $Coverage(I_{init}) = \{bl_1, \ldots, bl_n\}$ the n input blocks to be covered by input sets $I \subseteq I_{init}$ and we associate to each block bl_i an objective function $f_{bl_i}(.)$ defined by:

$$f_{bl_i}(I) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } bl_i \in Coverage(I) \\ \frac{1}{potential(I, bl_i)+1} & \text{otherwise} \end{cases}$$

The lower this value, the better. If bl_i is covered, then $f_{bl_i}(I) = 0$, otherwise $f_{bl_i}(I) > 0$. As expected, input sets that cover the objective are better than input sets that do not, and if two input sets do not cover the objective, then the normalized potential is used to break the tie.

3.4 Solutions to Our Problem

To solve our problem, we first need to retrieve, for each input $in \in I_{init}$ in the initial input set, 1) its cost $cost(in)$ without executing the considered MRs and 2) its coverage $Coverage(in)$, depending on the input characteristics that are relevant for vulnerability detection. Then, each element in the decision space is an input set $I \subseteq I_{init}$, which is associated with a *fitness vector*:

$$F(I) \stackrel{\text{def}}{=} [\omega(cost(I)), f_{bl_1}(I), \ldots, f_{bl_n}(I)]$$

Hence, we can define the Pareto front formed by the non-dominated solutions in our decision space (§ 2.3.1) and we formulate our problem definition as a many-objective optimization problem:

$$\underset{I \subseteq I_{init}}{\text{minimize}}\, F(I)$$

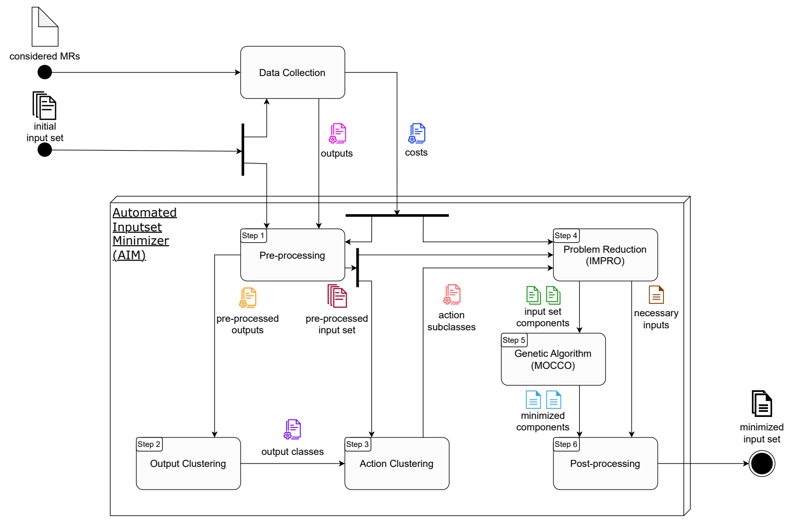


Fig. 2. Activity diagram of the Automated Input Minimizer (AIM) approach.

where the minimize notation means that we want to find or at least approximate the non-dominated decision vectors having a fitness vector on the Pareto front [34]. Because we want full input coverage (§ 3.1), the ultimate goal is a non-dominated solution I_{final} such that:

$$F(I_{final}) = [\omega(cost_{\min}), 0, \ldots, 0]$$

where $cost_{\min}$ is the cost of the cheapest subset of I_{init} with full input coverage.

4 OVERVIEW OF THE APPROACH

As stated in our problem definition (Section 3), we aim to reduce the cost of metamorphic security testing by minimizing an initial input set without removing inputs that are required to exercise distinct security vulnerabilities. To do so, we need to tackle the following sub-problems (§ 3.4):

(1) For each initial input, we need to determine its cost without executing the considered MRs.
(2) For each initial input, we need to determine its coverage. In the context of metamorphic testing for Web systems, we consider input blocks based on system outputs and input parameters.
(3) Amongst all potential input sets $I \subseteq I_{init}$, we search for a non-dominated solution I_{final} that preserves coverage while minimizing cost.

The *Automated Input Minimizer (AIM)* approach works in six steps whose workflow is depicted in Figure 2. AIM takes an *initial input set* and generates a *minimized input set*, i.e., a subset of the initial input set. The inputs in both sets can be used as source inputs for MST. We optimized AIM to work with test inputs for Web systems; in such context, each input is a sequence of actions (e.g., URL requests or interactions with the widget of a Web page) leading to Web pages as output. Such action sequences can be automated through Web testing frameworks such as Selenium [58].

AIM relies on analyzing the output and cost corresponding to each input. In the context of Web systems, we added a new feature to the MST-wi toolset to execute each input on the system and retrieve the content of the corresponding Web pages. Obtaining the outputs of the system is very inexpensive compared to executing the considered MRs. Moreover, to address our first sub-problem, we also updated MST-wi to retrieve the cost of an input without executing the considered MRs. We rely on a surrogate metric (i.e., the number of Web system interactions triggered by each source input when executed with MRs), which is both inexpensive and linearly correlated with execution time (§ 5.1).

In Step 1 (Pre-processing), AIM pre-processes the initial input set and the output information, by extracting relevant textual content from each returned Web page.

To address the second sub-problem, AIM relies on a *double-clustering* approach (Section 6), which is implemented by Step 2 (Output Clustering) and Step 3 (Action Clustering). For both steps, AIM relies on state-of-the-art clustering algorithms, which require to select hyper-parameter values (§ 6.1). *Output clustering* (§ 6.2) is performed on the pre-processed outputs, each generated cluster corresponding to an *output class*. Then, for each output class identified by the Output Clustering Step, *Action clustering* (§ 6.3) first determines the actions whose output belongs to the considered output class, then partitions these actions based on action parameters such as URL, username, and password, obtaining *action subclasses*. On the completion of Step 3, AIM has mapped each input to a set of action subclasses, used for measuring input coverage as per our problem definition.

Therefore, to preserve diversity in our input set, and especially to retain inputs that are required to exercise vulnerabilities, we require the minimized input set generated by AIM to cover the same action subclasses as the initial input set. That way, we increase the chances that the minimized input set contains at least one input able to exercise each vulnerability detectable with the initial input set.

Using cost and coverage information, AIM can address the last sub-problem. Since the size of the search space exponentially grows with the number of initial inputs, the solution cannot be obtained by exhaustive search. Actually, our problem is analogous to the knapsack problem [35], which is NP-hard, and is thus unlikely to be solved by deterministic algorithms. Therefore, AIM relies on metaheuristic search to find a solution (Step 5) after reducing the search space (Step 4) to the maximum extent possible.

In Step 4, since the search space might be large, AIM first reduces it to the maximal extent possible (Section 7) before resorting to metaheuristic search. To do so, it relies on the Inputset Minimization Problem Reduction Operator (IMPRO) component for problem reduction, which determine the necessary inputs, removes inputs that cannot be part of the solution, and partition the remaining inputs into input set *components* that can be independently minimized.

In Step 5, AIM applies a genetic algorithm (Section 8) to independently minimize each component. Because existing algorithms did not entirely fit our needs, we explain why we introduce MOCCO (Many-Objective Coverage and Cost Optimizer), a novel genetic algorithm based on two populations, one focusing in covering the objectives, and the other one on reducing cost. By restricting crossover so that one parent is selected in each population, we aim at converging towards a solution covering the objectives at a minimal cost, obtaining a *minimized input set component*.

Finally, after the genetic search is completed for each component, in Step 6 (Post-processing), AIM combines necessary inputs and inputs from the minimized components, in order to generate the *minimized input set* (Section 9).

Note that, even though in our study we focus on Web systems, steps 4 (IMPRO), 5 (MOCCO), and 6 (post-processing), which form the core of our solution, are generic and can be applied to any system. Moreover, step 1 (pre-processing) and steps 2 and 3 (double-clustering) can be tailored to apply AIM to other domains (e.g., desktop applications, embedded systems). And though we relied on MST-wi to collect our data, AIM does not depend on a particular data collector, and using or implementing another data collector would enable the use of our approach in other contexts.

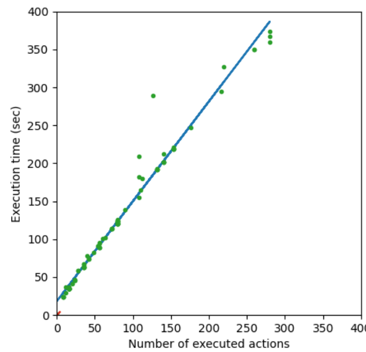


Fig. 3. Linear regression between the number of executed actions and the execution time of a metamorphic relation. Each input is represented by a green dot, while the blue line depicts the linear regression model.

5 STEP 1: DATA COLLECTION AND PRE-PROCESSING

To address our first sub-problem (Section 4), we determine the cost of each initial input (§ 5.1). Then, since in this study we focus on Web systems, we detail how we extract meaningful textual content from the Web pages obtained with the initial inputs (§ 5.2).

5.1 Input Cost

We want to reduce the time required to execute MRs by minimizing the total cost of the initial input set (§ 3.4). However, knowing the execution time of each initial input would require to execute them on the considered MRs, hence defeating the purpose of input set minimization. Thus, to determine an input cost, we rely on a surrogate metric for execution time.

In the context of a Web system, inputs are sequences of actions. An MR generates follow-up inputs by modifying source inputs (e.g., changing the user or the URL). Given the potentially large number of actions in an input and the diversity of parameter values for each action, the number of follow-up inputs may be large. Then, source and follow-up inputs are executed on the system to verify security properties, potentially resulting in a large execution time required. Fortunately, the execution time of an MR with an input tends to be linearly correlated with the number of executed actions from source and follow-up inputs, as illustrated in Example 3 below.

Example 3. We investigate here one MR written for CWE_863 [42] and randomly selected initial inputs. We executed the MR for each input and measured its execution time and the number of executed actions from source and follow-up inputs. We performed a linear regression between execution time and actions, represented by the blue line depicted in Figure 3. The coefficient of determination is 0.978, indicating a strong linear correlation between MR execution time and the number of executed actions.

Note that counting the number of actions to be executed is inexpensive compared to executing them on the system, then checking for the verdict of the output relation. For instance, counting the number of actions to be executed for eleven MRs with Jenkins' initial input set (Section 10) took less than five minutes, while executing the MRs took days. Hence, the number $nbActions(mr, in)$ of actions to be executed by mr using input in can be used as a surrogate metric for execution time.

We thus define the cost of an input as follows:

$$cost(in) \stackrel{\text{def}}{=} \sum_{mr \in MRs} nbActions(mr, in)$$

When $cost(in) = 0$, input *in* was not exercised by any MR due to the preconditions in these MRs. Hence, *in* is not useful for MST and can be removed without loss from the initial input set.

5.2 Output Representation

Since, in this study, we focus on Web systems, the outputs of the system are Web pages. Fortunately, collecting these pages using a crawler and extracting their textual content is inexpensive compared to executing MRs. Hence, we can use system outputs to determine relevant input blocks (Section 3).

We focus on textual content extracted from the Web pages returned by the Web system under test. We remove from the textual content of each Web page all the data that is shared among many Web pages and thus cannot characterize a specific page, like system version, date, or (when present) the menu of the Web page. Moreover, to focus on the meaning of the Web page, we consider the remaining textual content not as a string of characters but as a sequence of words. Also, following standard practice in natural language processing, we apply a stemming algorithm to simplify words to their simplest form, thus considering distinct words with the same stem as equivalent, for instance the singular and plural forms of the same word. Finally, we remove stopwords, numbers, and special characters, in order to focus on essential textual information.

6 STEPS 2 AND 3: DOUBLE CLUSTERING

We want to minimize an initial input set in order to reduce the cost of MST while preserving at least one input able to exercise each vulnerability affecting the SUT; of course, in practice, such vulnerabilities are not known in advance but should be discovered by MST. Hence, we have to determine in which cases two inputs are distinct enough so that both should be kept in the minimized input set, and in which cases some inputs are redundant with the ones we already selected and thus can be removed.

Since, for practical reasons, we want to avoid making assumptions regarding the nature of the Web system under test (e.g., programming language or underlying middleware), we propose a black-box approach relying on input and output information to determine which inputs we have to keep or remove. To determine which inputs are similar and which significantly differ, we rely on clustering algorithms. Precisely, we rely on the K-means, DBSCAN, and HDBSCAN algorithms to cluster our data points. Each of them has a set of hyper-parameters to be set and we first detail how these hyper-parameters are obtained using *Silhouette analysis* (§ 6.1).

In the context of a Web system, each input is a sequence of actions, each action enabling a user to access a Web page, and each output is a Web page. Furthermore, each action uses a request method, either a POST or GET method, to send an HTTP request to the server. After gathering output and action information, we perform *double-clustering* on our data points, i.e., two clustering steps performed in sequence:

(1) *Output clustering* (§ 6.2) uses the outputs of the Web system under test, i.e., textual data obtained by pre-processing content from Web pages (§ 5.2). We define an output distance (§ 6.2.1) to quantify similarity between these outputs that is used by AIM to run Silhouette analysis and clustering algorithms to partition outputs into *output classes* (§ 6.2.2).
(2) *Action clustering* (§ 6.3) then determines input coverage. First, AIM puts into the same *action set* all the actions leading to outputs in the same output class (§ 6.3.1). Then, AIM refines each action set by partitioning the actions it contains using action parameters. To do so, it first uses the request method (§ 6.3.2) to split action sets into parts. Then, we define an

action distance (§ 6.3.3) based on the URL (§ 6.3.4) and other parameters (§ 6.3.5) of the considered actions. Finally, AIM relies on Silhouette analysis and clustering algorithms to partition each part of an action set into *action subclasses* (§ 6.3.6), defining our input blocks (Section 3).

6.1 Hyper-parameters Selection

In this study, we rely on the common K-means [4], DBSCAN [21], and HDBSCAN [39] clustering algorithms to determine both the output classes (§ 6.2) and the action subclasses (§ 6.3). These clustering algorithms require a few hyper-parameters to be set. For K-means, one needs to select the number of clusters k. For DBSCAN, one need to select the distance threshold ϵ to determine the ϵ-neighbourhood of each data point and the minimum number of neighbours n needed for a data point to be considered a core point. For HDBSCAN, one needs to select the minimum number n of individuals required to form a cluster.

To select the best values for these hyper-parameters, we rely on *Silhouette analysis*. Though the Silhouette score is a common metric used to determine optimal values for hyper-parameters [5, 6], it is obtained from the average Silhouette score of the considered data points. Thus, for instance, clusters with all data points having a medium Silhouette score cannot be distinguished from clusters where some data points have a very large Silhouette score while others have a very small one. Hence, having a large Silhouette score does not guarantee that all the data points are well-matched to their cluster. To quantify the variability in the distribution of Silhouette scores, we use Gini index, a common measure of statistical dispersion. If the Gini index is close to 0, then Silhouette scores are almost equal. With a Gini index is close to 1, then variability in Silhouette score across data points is high.

Hence, for our Silhouette analysis, we consider two objectives: Silhouette score and the Gini index of the Silhouette scores. The selection of hyper-parameters is therefore a multi-objective problem with two objectives. We rely on the common NSGA-II evolutionary algorithm [19] to solve this problem and approximate the Pareto front regarding both Silhouette score and Gini index. Then, we select the item in the Pareto front that has the highest Silhouette score.

6.2 Step 2: Output Clustering

Output clustering consists in defining an output distance (§ 6.2.1) to quantify dissimilarities between Web system outputs, and then to partition the outputs to obtain *output classes* (§ 6.2.2).

A user communicates with a Web system using actions. Hence, an input for a Web system is a sequence of actions (e.g., login, access to a Web page, logout). The *length* of an input in is its number of actions and is denoted by $len(in)$. As the same action may occur several times in an input, a given occurrence of an action is identified by its position in the input. If $1 \leq i \leq len(in)$, we denote by $action(in, i)$ the action at position i in input in. Outputs of a Web system are textual data obtained by pre-processing the content from Web pages (§ 5.2). The accessed Web page depends not only on the considered action, but also on the previous ones; for instance if the user has logged into the system. Hence, we denote by $output(in, i)$ the output of the action at position i in in.

6.2.1 Output distance. In this study, we use system outputs (i.e., Web pages) to characterize system states. Hence, two actions that do not lead to the same output should be considered distinct because they bring the system into different states. More generally, dissimilarity between outputs is quantified using an *output distance*. Since we deal with textual data, we consider both Levenshtein and bag distances. Levenshtein distance is usually a good representation of the difference between two textual contents [26, 63]. However, computing the minimal number of edits between two strings can be costly, since the complexity of the Levenshtein distance between two strings is $O(len(s_1) \times len(s_2))$, where $len(.)$ is the length of the string [68]. Thus, we consider the bag distance [40] as an alternative to the Levenshtein distance, because its complexity is only $O(len(s1) + len(s_2))$ [9]. But it does not take into account the order of words and is thus less precise than Levenshtein distance.

6.2.2 Output Classes. Since we want to minimize an input set while preserving input diversity, for each output obtained by executing the initial input set, we want in the minimized input set at least one input that lead to this output. To do so, we partition the textual content we obtained from Web pages (§ 5.2) using the K-means, DBSCAN, and HDBSCAN clustering algorithms, setting the hyper-parameters using Silhouette analysis (§ 6.1), and determining similarities between outputs using the chosen output distance. We call *output classes* the obtained clusters and we denote by $OutputClass(in, i)$ the unique output class $output(in, i)$ belongs to.

6.3 Step 3: Action Clustering

Exercising all the Web pages is not sufficient to discover all the vulnerabilities; indeed, vulnerabilities might be detected through specific combinations of parameter values associated to an action (e.g., values belonging to a submitted form). Precisely, actions on a Web system can differ with respect to a number of *parameters* that include the URL (allowing the action to perform a request to a Web server), the method of sending a request to the server (like GET or POST), URL parameters (e.g., `http://mydomain.com/myPage?urlParameter1=value1&urlParameter2=value`), and entries in form inputs (i.e., textarea, textbox, options in select items, datalists).

Based on the obtained output classes, *action clustering* first determines *action sets* (§ 6.3.1) such that actions leading to outputs in the same output class are in the same action set. Then, action clustering refines each action set by partitioning the actions it contains using actions parameters. First, we give priority to the method used to send a request to the server, so we split each action set using the request method (§ 6.3.2). Then, to quantify the dissimilarity between two actions, we define an action distance (§ 6.3.3) based on the URL (§ 6.3.4) and other parameters (§ 6.3.5) of the considered actions. That way, action clustering refines each action set into *action subclasses* (§ 6.3.6).

6.3.1 Action Sets. Based on the obtained output classes (§ 6.2.2), AIM determines *action sets* such that actions leading to outputs in the same output class *outCl* are in the same action set:

$$ActionSet(outCl) \stackrel{\text{def}}{=} \{act \mid \exists in, i : action(in, i) = act \land OutputClass(in, i) = outCl\}$$

Note that, because an action can have different outputs depending on the considered input, it is possible for an action to belong to several actions sets, corresponding to several output classes.

6.3.2 Request Partition. Each action uses either a POST or GET method to send an HTTP request to the server. Actions (such as login) that send the parameters to the server in the message body use the POST method, while actions that send the parameters through the URL use the GET method. As this difference is meaningful for distinguishing different action types, we split each action set into two parts: the actions using a POST method and those using a GET method.

6.3.3 Action Distance. After the request partition (§ 6.3.2), we consider one part of an action set at a time and we refine it using an action distance quantifying dissimilarity between actions based on remaining parameters (i.,e., URL, URL parameters, form entries). In the context of a Web system, each Web content is identified by its URL, so we give more importance to this parameter. We denote $\mathrm{url}(act_i)$ the URL of action act_i. For the sake of clarity, we call in the rest of the section *residual parameters* the parameters of an action which are not its request method or its URL and we denote $\mathrm{resids}(act_i)$ the residual parameters of action act_i. Since we give more importance to the URL, we represent the distance between two actions by a real value, where the integral part corresponds to the distance between their respective URLs and the decimal part to the distance between their respective residual parameters:

$$\text{actionDist}(act_1, act_2) \stackrel{\text{def}}{=} \text{urlDist}(\text{url}(act_1), \text{url}(act_2)) + \text{paramDist}(\text{resids}(act_1), \text{resids}(act_2))$$

where the URL distance urlDist(., .) is defined in § 6.3.4 and returns an integer, and the parameter distance paramDist(., .) is defined in § 6.3.5 and returns a real number between 0 and 1.

6.3.4 URL distance. A URL is represented as a sequence of at least two words, separated by ':// ' between the first and second word, then by '/ ' between any other words. The length of a URL *url* is its number of words, denoted $len(url)$. Given two URLs, url_1 and url_2, their *lowest common ancestor* is the longest prefix they have in common, denoted $\text{LCA}(url_1, url_2)$. We define the distance between two URLs as the total number of words separating them from their lowest common ancestor:

$$\text{urlDist}(url_1, url_2) \stackrel{\text{def}}{=} len(url_1) + len(url_2) - 2 \times len(\text{LCA}(url_1, url_2))$$

Example 4. We consider two URLs: url_1 = `http://hostname/login` and url_2 = `http://hostname/job/try1/lastBuild`. Their LCA is `http://hostname`. There is one word (`login`) separating url_1 from their LCA, hence $len(url_1) - len(\text{LCA}(url_1, url_2)) = 1$. Moreover, there are three word (`job`, `try1`, and `lastBuild`) separating url_2 from their LCA, hence $len(url_2) - len(\text{LCA}(url_1, url_2)) = 3$. Therefore, the URL distance between url_1 and url_2 is $\text{urlDist}(url_1, url_2) = 1 + 3 = 4$.

6.3.5 Parameter Distance. To quantify the dissimilarity between residual parameters, we first independently quantify the dissimilarity between pairs of parameters of the same type. Since, in our context, we exercise vulnerabilities by only using string or numerical values, we ignore parameter values of other types such as byte arrays (e.g., an image uploaded to the SUT). In other contexts, new parameter distance functions suited to other input types may be required. For strings, we use the Levenshtein distance [26, 63], whereas for numerical values we consider the absolute value of their difference [12]:

$$\text{paramValDist}(v_1, v_2) \stackrel{\text{def}}{=} \begin{cases} LevenshteinDist(v_1, v_2) & \text{if type}(v_1) = \text{str} = \text{type}(v_2) \\ |v_1 - v_2| & \text{if type}(v_1) = \text{int} = \text{type}(v_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Since we have parameters of different types, we normalize the parameter distance using the normalization function $\omega(x) = \frac{x}{x+1}$ (§ 3.3). Then, we add these normalized distances together, and normalize the sum to obtain a result between 0 and 1. We compute the parameter distance in case of *matching parameters*, i.e., the number of parameters is the same and the corresponding parameters have the same type. Otherwise, we assume the largest distance possible, which is 1 due to the normalization. This is the only case where the value 1 is reached, as distance lies otherwise in [0 1[, as expected for a decimal part (§ 6.3.3):

$$\text{paramDist}(resids_1, resids_2) \stackrel{\text{def}}{=} \begin{cases} \omega\Big(\sum\limits_{0 \le i < len(resids_1)} \omega(\text{paramValDist}(resids_1^{[i]}, resids_2^{[i]}))\Big) \\ \quad \text{if } resids_1 \text{ and } resids_2 \text{ have matching parameters} \\ 1 \text{ otherwise} \end{cases}$$

where $resids_1 = \text{resids}(act_1)$, $resids_2 = \text{resids}(act_2)$, and $resids^{[i]}$ is the i-th element of *resids*.

Example 5. We consider two actions act_1 and act_2 having matching parameters with the following values for page number, username, and password:

$$\begin{aligned} resids_1 &= [\text{pageNb: int} = 10, \text{username: str} = \text{“John ”}, \text{password: str} = \text{“qwerty ”}] \\ resids_2 &= [\text{pageNb: int} = 42, \text{username: str} = \text{“Johnny ”}, \text{password: str} = \text{“qwertyuiop ”}] \end{aligned}$$

The distance for the page number is $\text{paramValDist}(10, 42) = 32$, which is normalized into $\frac{32}{32+1} \approx 0.97$. The distance for the username is $\text{paramValDist}(\text{“John ”}, \text{“Johnny ”}) = 2$, normalized into $\frac{2}{2+1} \approx 0.66$. The distance for the password is $\text{paramValDist}(\text{“qwerty ”}, \text{“qwertyuiop ”}) = 4$, normalized into $\frac{4}{4+1} = 0.80$. Thus, the parameter distance is $\text{paramDist}(\mathit{resids}_1, \mathit{resids}_2) \approx \frac{0.97+0.66+0.80}{0.97+0.66+0.80+1} \approx 0.71$.

6.3.6 Action Subclasses. We partition both parts of each action set (§ 6.3.2) using the K-means, DBSCAN, or HDBSCAN clustering algorithms, setting the hyper-parameters using our Silhouette analysis (§ 6.1), and quantifying action dissimilarity using our action distance (§ 6.3.3), obtaining clusters we call *action subclasses*. We denote by $\mathit{ActionSubclass}(\mathit{act}, \mathit{actSet})$ the unique action subclass bl from the action set actSet such that $\mathit{act} \in \mathit{bl}$. For the sake of simplicity, we denote $\mathit{Subclass}(\mathit{in}, i)$ the action subclass corresponding to the i-th action in input in:

$$\mathit{Subclass}(\mathit{in}, i) \stackrel{\text{def}}{=} \mathit{ActionSubclass}(\mathit{action}(\mathit{in}, i), \mathit{ActionSet}(\mathit{OutputClass}(\mathit{in}, i)))$$

Finally, in our study, the objectives covered by an input in are:

$$\mathit{Coverage}(\mathit{in}) \stackrel{\text{def}}{=} \{\mathit{Subclass}(\mathit{in}, i) \mid 1 \leq i \leq \mathit{len}(\mathit{in})\}$$

7 STEP 4: PROBLEM REDUCTION

The search space for our problem (§ 3.4) consists of all the subsets of the initial input set, which leads to 2^m potential solutions, with m being the number of initial inputs.

For this reason, AIM integrates a *problem reduction* step, implemented by the Inputset Minimization Problem Reduction Operator (IMPRO) component, to minimize the search space before solving the search problem in the next step (Section 8). We apply the following techniques to reduce the size of the search space:

- *Determining redundancy*: Necessary inputs (§ 3.3) must be part of the solution, hence one can only investigate redundant inputs (§ 7.3). Moreover, one can restrict the search by removing the objectives already covered by necessary inputs. Finally, if a redundant input does not cover any of the remaining objectives, it will not contribute to the final coverage, and hence can be removed.
- *Removing duplicates*: Several inputs may have the same cost and coverage. In this case, we consider them as *duplicates* (§ 7.4). Thus, we keep only one and we remove the others.
- *Removing locally-dominated inputs*: For each input, if there exists other inputs that cover the same objectives at a same or lower cost, then the considered input is *locally-dominated* by the other inputs (§ 7.5) and is removed.
- *Dividing the problem*: We consider two inputs covering a common objective as being connected. Using this relation, we partition the search space into connected *components* that can be independently solved (§ 7.6), thus reducing the number of objectives and inputs to investigate at a time.

Before detailing these techniques, we first explain in which order there are applied (§ 7.1).

7.1 Order of Reduction Techniques

We want to perform first the least expensive reduction techniques, to sequentially reduce the cost of the following more expensive techniques. Determining redundancy requires $O(m \times c)$ steps, removing duplicates requires $O(m^2)$ steps, and removing locally-dominated inputs requires $O(m \times 2^n)$ steps, where m is the number of inputs, c is the maximal number of objectives covered by an input, and n is the maximal number of neighbors for an input (i.e., the number of other inputs that cover an objective shared with the considered input). In our study, we assume $c < m$. Hence, we first determine redundancy, then remove duplicates, and remove locally-dominated inputs. Dividing the problem requires exploring neighbors and comparing non-visited inputs with visited ones, so it is potentially the most costly of these reduction techniques; hence, it is performed at the end.

After determining redundancy, the removal of already covered objectives may lead to new inputs being duplicates or locally-dominated. Moreover, the removal of duplicates or locally-dominated inputs may lead to changes in redundancy, making some previously redundant inputs necessary. Hence, these reduction techniques should be iteratively applied, until a stable output is reached. Such output can be detected by checking if inputs were removed during an iteration. Indeed, when no input is removed then, during the next iteration, no new input becomes necessary, so the objectives to be covered do not change, and thus there is no opportunity to remove more inputs.

Therefore, the order is as follows. We first initialize variables (§ 7.2). Then we repeat, until no input is removed, the following steps: determine redundancy (§ 7.3), remove duplicates (§ 7.4), and remove locally-dominated inputs (§ 7.5). Finally, we divide the problem into sub-problems (§ 7.6).

7.2 Initializing Variables

During problem reduction, we consider three variables: I_{necess}, the set of the inputs that has to be part of the final solution, I_{search}, the remaining inputs to be investigated, and $Coverage_{obj}$, the objectives that remain to be covered by subsets of I_{search}. I_{necess} is initially empty. I_{search} is initialized as the initial input set. $Coverage_{obj}$ is initialized as the coverage of the initial input set (§ 6.3.6 and § 3.1).

7.3 Determining Redundancy

Each time this technique is repeated, the redundancy of the remaining inputs is computed. Among them, inputs which are necessary (§ 3.3) in I_{search} for the objectives in $Coverage_{obj}$ have to be included in the final solution, otherwise some objectives will not be covered:

$$I_{necess} \leftarrow I_{necess} \cup I_{necess}^{new}$$

where $I_{necess}^{new} = I_{search} \setminus Redundant(I_{search})$.

Then, the objectives already covered by the necessary inputs are removed:

$$Coverage_{obj} \leftarrow Coverage_{obj}^{new}$$

where $Coverage_{obj}^{new} = Coverage(I_{search}) \setminus Coverage(I_{necess}^{new})$. Hence, in the following, we consider for each remaining input $in \in I_{search}$ only their coverage regarding the remaining objectives, i.e., $Coverage(in) \cap Coverage_{obj}$, instead of $Coverage(in)$.

Finally, some redundant inputs may cover only objectives that are already covered by necessary inputs. In that case, they cannot be part of the final solution because they would contribute to the cost but not to the coverage of the objectives. Hence, we restrict without loss the search space for our problem by considering only redundant inputs that can cover the remaining objectives:

$$I_{search} \leftarrow \{in \in Redundant(I_{search}) \mid Coverage(in) \cap Coverage_{obj}^{new} \neq \varnothing\}$$

7.4 Removing Duplicates

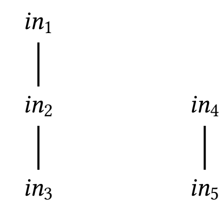
In the many-objective problem described in § 3.3, inputs are characterized by their coverage (§ 6.3.6) and their cost (§ 5.1). Hence, two inputs are considered equivalent if they have the same coverage and cost. For each equivalence class containing at least two inputs, we say these inputs are *duplicates*. IMPRO selects one representative per equivalence class in I_{search} and removes the others.

7.5 Removing Locally-dominated Inputs

For a given input $in \in I_{search}$, if the same coverage can be achieved by one or several other input(s) for at most the same cost, then in is not required for the solution. Formally, we say that

	bl_1	bl_2	bl_3	bl_4	bl_5	bl_6	bl_7
in_1	✓	✓					
in_2		✓	✓				
in_3			✓	✓			
in_4					✓	✓	
in_5						✓	✓

(a) Coverage Matrix



(b) Overlapping Graph

Fig. 4. Inputs covering one objective in common (left) are connected in the overlapping graph (right).

the input $in \in I_{search}$ is *locally dominated* by the subset $S \subseteq I_{search}$, denoted $in \sqsubseteq S$, if $in \notin S$, $Coverage(in) \subseteq Coverage(S)$, and $cost(in) \geq cost(S)$. In order to simplify the problem, inputs that are locally dominated should be removed from the remaining inputs I_{search}.

Removing a redundant input in can only affect the redundancy of the inputs in I that cover objectives in $Coverage(in)$ (§ 3.3). Hence, we consider two inputs as being connected if they cover at least one common objective. Formally, we say that two inputs in_1 and in_2 *overlap*, denoted by $in_1 \sqcap in_2$, if $Coverage(in_1) \cap Coverage(in_2) \neq \varnothing$. The name of the *local* dominance relation comes from the fact proved in the appendix (Proposition 3) that, to determine if an input is locally-dominated, one has only to check amongst its neighbors for the overlapping relation instead of amongst all the remaining inputs. Formally, if $in_1 \sqsubseteq S$, then $in_1 \sqsubseteq S \cap \{in_2 \in I_{search} \mid in_1 \sqcap in_2\}$.

One concern is that removing a locally-dominated input could alter the local dominance of other inputs. Hence, like reduction steps (§ 3.3), IMPRO would have to take into account the order for removing locally-dominated inputs. Fortunately, this is not the case for local dominance. We prove in the appendix (Theorem 2) that, for every locally-dominated input $in \in I_{search}$, there always exists a subset $S \subseteq I_{search}$ of not locally-dominated inputs such that $in \sqsubseteq S$. Hence, the locally dominated inputs can be removed in any order without reducing coverage or preventing cost reduction, both being ensured by non locally-dominated inputs. Therefore, IMPRO keeps in the search space only the remaining inputs that are not locally-dominated:

$$I_{search} \longleftarrow \{in \in I_{search} \mid \forall S \subseteq I_{search} : in \not\sqsubseteq S\}$$

7.6 Dividing the Problem

After removing as many inputs as possible, we leverage the overlapping relation (§ 7.5) to partition the remaining inputs into independent components, in a divide-and-conquer approach. Formally, for a given input set I, we denote $\mathcal{G}_{\sqcap}(I) \stackrel{\text{def}}{=} [\mathcal{V}(I), \mathcal{E}(I)]$ its *overlapping graph*, i.e., the undirected graph such that vertices $\mathcal{V}(I) \stackrel{\text{def}}{=} I$ are inputs in I and edges $\mathcal{E}(I) \stackrel{\text{def}}{=} \{\{in_1, in_2\} \mid in_1, in_2 \in \mathcal{V}(I) \wedge in_1 \sqcap in_2\}$ correspond to the overlapping relation, and $OverlapComps(I) \stackrel{\text{def}}{=} Components(\mathcal{G}_{\sqcap}(I))$ the set of the *overlapping components* of I, where $Components(.)$ denotes the set of connected components.

Example 6. In Figure 4, we represent on the left (Figure 4a) the input blocks covered by each input in the search space and the corresponding overlapping graph on the right (Figure 4b). The overlapping components of the graph are $\{in_1, in_2, in_3\}$ and $\{in_4, in_5\}$.

Such overlapping components are important because we prove in the appendix (Proposition 8) that inputs in an overlapping component can be removed without altering the redundancy of inputs in other overlapping components. Formally, if $C_1, \dots, C_c \in OverlapComps(I)$ denote c overlapping components of I and, in each component C_i, there exists a valid order of removal steps $[in^i_1, \dots, in^i_{n_i}]$, then the concatenation $[in^1_1, \dots, in^1_{n_1}] + \cdots + [in^c_1, \dots, in^c_{n_c}]$ is a valid order of removal steps in I.

Similarly, we prove in the appendix (Theorem 3) that the gain, i.e., the maximal cost reduction from removing redundant inputs (§ 3.3), can be independently computed on each overlapping component, i.e., for each input set I, $gain(I) = \sum_{C \in OverlapComps(I)} gain(C)$.

Hence, instead of searching a solution on I_{search} to solve our initial problem (§ 3.4), we use a divide-and-conquer strategy to split the problem into more manageable sub-problems that can be independently solved on each overlapping component $C \in OverlapComps(I_{search})$. We denote $Coverage_{obj}(C) \stackrel{\text{def}}{=} Coverage_{obj} \cap Coverage(C)$ the remaining objectives to be covered by inputs in C and we formulate the sub-problem on the overlapping component similarly to the initial problem:

$$\underset{I \subseteq C}{\text{minimize}}\, F_C(I) \stackrel{\text{def}}{=} [\omega(cost(I)), f_{bl_1}(I), \ldots, f_{bl_n}(I)]$$

where $Coverage_{obj}(C) = \{bl_1, \ldots, bl_n\}$ and the minimize notation as detailed in § 2.3.1. We denote I_C a non-dominated solution with full coverage, such that $F_C(I_C) = [\omega(cost_{\min}), 0, \ldots, 0]$.

8 STEP 5: GENETIC SEARCH

Obtaining an optimal solution I_C to our sub-problem (§ 7.6) on an overlapping component C would be similar to solving the knapsack problem, which is NP-hard [35]. To be more precise, our problem is equivalent to the 0-1 knapsack problem, which consists in selecting a subset of items to maximize a total value, while satisfying a weight capacity. More specifically, since we consider a many-objective problem (§ 3.2), we must address the multidimensional variant of the 0-1 knapsack problem, where each item has many "weights", one per considered objective. In our case, we minimize the total cost instead of maximizing total value and ensure the coverage of each action objective instead of making sure that each weight capacity is not exceeded. Furthermore, the 0-1 multidimensional knapsack problem is harder than the initial knapsack problem as it does not admit a fully polynomial time approximation scheme [33], hence the need for a meta-heuristic.

For our approach to scale, we adopt a genetic algorithm because it is known to find good approximations in reasonable execution time [54] and has been widely used in software testing. An input set $I \subseteq C$ can thus be seen as a chromosome, where each gene corresponds to an input $in \in C$, the gene value being 1 if $in \in I$ and 0 otherwise. Though several many-objective algorithms have been successfully applied within the software engineering community, like NSGA-III [18, 29] and MOSA [54] (§ 2.3.2), these algorithms do no entirely fit our needs (§ 8.1). Hence, we propose MOCCO (Many-Objective Coverage and Cost Optimizer), a novel genetic algorithm based on two populations. We first explain how these populations are initialized (§ 8.2). Then, for each generation, MOCCO performs the following standard steps: selection of the parents (§ 8.3), crossover of the parents to produce an offspring (§ 8.4), mutation of the offspring (§ 8.5), and update of the populations (§ 8.6) to obtain the next generation. The process continues until a termination criterion is met (§ 8.7). Then, we detail how MOCCO determines the solution I_C to each sub-problem.

8.1 Motivation for a Novel Genetic Algorithm

While our problem is similar to the multidimensional 0-1 knapsack problem, it is not exactly equivalent, since standard solutions to the multidimensional knapsack problem have to ensure that, for each "weight type", the total weight of the items in the knapsack is below weight capacity, while we want to ensure that, for each objective, at least one input in the minimized input set covers it. Hence, standard solutions based on genetic search are not applicable in our case, and we focus on genetic algorithms able to solve many-objective problems in the context of test case generation or minimization. We have explained earlier (§ 2.3.2) the challenges raised by many-objective problems and how the NSGA-III [18, 29] and MOSA [54] genetic algorithms tackle such challenges.

NSGA-III has the advantage of letting users choose the parts of the Pareto front they are interested in, by providing reference points. Otherwise, it relies on a systematic approach to place points on a normalized hyperplane. While this approach is useful in general, this is not what we want to achieve. Indeed, we are interested only in solutions that are close to a utopia point $[0, 0, \ldots, 0]$ (§ 3.3) covering every objective at no cost. Hence, we do not care about diversity over the Pareto front, and we want to explore a very specific region of the search space. Moreover, apart from the starting points, the main use of the reference points in NSGA-III is to determine, after the first nondomination fronts are obtained from NSGA-II [19], the individuals to be selected from the last considered front, so that the population reaches a predefined number of individuals. This is not a problem we face because we know there is only one point (or several points, but at the same coordinates) in the Pareto front that would satisfy our constraint of full coverage at minimal cost. Hence, we do not use the Pareto front as a set of solutions, even if we intend to use individuals in the Pareto front as intermediate steps to reach relevant solutions while avoiding getting stuck in a local optimum and providing pressure for reducing the cost of the candidates.

Regarding MOSA [54], trade-offs obtained from approximating the Pareto front are only used for maintaining diversity during the search, which is similar to what we intend to do. But, as opposed to the use case tackled by MOSA, in our case determining inputs covering a given objective is straightforward. Indeed, each objective is covered by at least one input from the initial input set, and for each objective we can easily determine inputs that are able to cover it (§ 3.2). Hence, individuals ensuring the coverage of the objectives are easy to obtain, while the hard part of our problem is to determine a combination of inputs able to cover all the objectives at a minimal cost. Hence, even if MOSA may find a reasonable solution, because it focuses on inputs individually covering an objective and not on their collective coverage and cost, it is unlikely to find the best solution.

Because common many-objective algorithms for software engineering problems do not fit our needs, we propose a novel genetic algorithm named MOCCO. We take inspiration from MOSA [54] by considering two populations: 1) a population of solutions (like MOSA's archive), called the *roofers* because they cover all the objectives (§ 6.3.6), and 2) a population of individuals on the Pareto front (§ 3.4), called the *misers* because they minimize the cost, while not covering all objectives.

To address challenge 1 (§ 2.3.2), we take inspiration from the whole suite approach [23] which counts covered branches as a single objective, by defining the *exposure* as the sum of the coverage objective functions (§ 3.3):

$$exposure(I) \stackrel{\text{def}}{=} \sum_{bl_i \in Coverage_{obj}(C)} f_{bl_i}(I)$$

Since $f_{bl_i}(.)$ is zero when the objective bl_i is covered, the larger the exposure, the less the input set coverage. As described in § 3.2, we do not the exposure as objective because we want to distinguish between input blocks. But we use it as a weight when randomly selecting a parent amongst the misers (§ 8.3), so that the further away a miser is from complete coverage, the less likely it is to be selected. That way, we aim to benefit from the large number of dimensions to avoid getting stuck in a local optimum and to have a better convergence rate [34], while still focusing the search on the region of interest.

Since we want to deeply explore this particular region, we do not need to preserve diversity over the whole Pareto front. Therefore, we do not use diversity operators, avoiding challenge 2 (§ 2.3.2).

Finally, we address challenge 3 (§ 2.3.2) by 1) restricting the recombination operations and 2) tailoring them to our problem.

(1) A crossover between roofers can only happen during the first generations, when no miser is present in the population. After the first miser is generated, crossover (§ 8.4) is allowed

only between a roofer and a miser. Hence, the roofer parent provides full coverage while the miser parent provides almost full coverage at low cost. Moreover, because of how the objective functions are computed (§ 3.3), the not-yet-covered objectives are likely to be covered in an efficient way. That way, we hope to increase our chances of obtaining offspring with both large coverage and low cost.

(2) Not only our recombination strategy is designed to be computationally efficient (by minimizing the number of introduced redundancies), but we exploit our knowledge of input coverage to determine a meaningful crossover between parents, with inputs from one parent for one half of the objectives and inputs from the other parent for the other half.

8.2 Population Initialization

During the search, because we need diversity to explore the search space, we consider a population, with a size (n_{size}) greater than one, of the least costly individuals generated so far that satisfy full coverage. We call *roofers* such individuals, by analogy with covering a roof, and we denote $Roofers(n)$ the roofer population at generation n.

But focusing only on the roofers would prevent us to exploit the least expensive solutions obtained in the Pareto front while trying to minimize the overlapping component (§ 7.6). Instead, inputs that do not cover all the objectives, and will thus not be retained for the final solution, but are efficient at minimizing cost, are thus useful as intermediary steps towards finding an efficient solution. Hence, we maintain a second population, formed by individuals that are non-dominated so far and minimize cost while not covering all the objectives. We call *misers* such individuals, because they focus on cost reduction more than objective coverage, and we denote $Misers(n)$ the miser population at generation n.

The reason for maintaining two distinct populations is to restrict the crossover strategy (§ 8.4) so that (in most cases) one parent is a roofer and one parent is a miser. Since misers prioritize cost over coverage, a crossover with a miser tends to reduce cost. Because roofers prioritize coverage over cost, a crossover with a roofer tends to increase coverage. Hence, with such a strategy, we intend to converge towards a solution minimizing cost and maximizing coverage.

For both populations, we want to ensure that the individuals are reduced (§ 3.3), i.e., they contain no redundant inputs. Hence, during the initialization and updates of these populations, we ensure that removal steps are performed. Because, as detailed in the following, the number of redundant inputs obtained for each generation is small, the optimal order of removal steps can be exhaustively computed. We denote by $reduc(I)$ the input set I after these removal steps. This limits the exploration space, by avoiding parts of the search space that are not relevant. Indeed, non-reduced input sets are likely to have a large cost and hence to be far away for the utopia point of full coverage at no cost we intend to focus on. Considering such non-reduced input sets would consist in exploring the search space by over-covering the objectives, while with the misers we explore it by under-covering it, thus focusing the search on the region of interest.

The miser population is initially empty, i.e., $Misers(0) \stackrel{\text{def}}{=} \varnothing$, as misers are generated during the search through mutations (§ 8.6). We now detail how the roofer population $Roofers(0)$ is initialized. First, the population is empty $Roofers(0) = \varnothing$. Then, we repeat the following process until the initial set of roofers reaches the desired number of individuals, i.e., $\text{card}(Roofers(0)) = n_{size}$. We start with an empty individual $I = \varnothing$. We repeat the following steps until all the objectives in $Coverage_{obj}(C)$ have been covered by I.

(1) We randomly select an uncovered objective $bl \in Coverage_{obj}(C) \setminus Coverage(I)$ using a uniform distribution.

(2) We randomly select an input $in_1 \in Inputs(bl)$ able to cover such objective, using the following distribution:

$$\mathrm{P}_{init}(in_1) \stackrel{\text{def}}{=} \frac{\frac{1}{1+occurrence(in_1)}}{\sum_{in_2 \in Inputs(bl)} \frac{1}{1+occurrence(in_2)}}$$

where $occurrence(in)$ denotes the number of times the input in was selected in $Roofers(0)$ so far. Indeed, if an input is not present in both populations, the only way to generate it is through random mutations (§ 8.5). Hence, to be able to explore the results of most inputs during the search, it is beneficial to have a diversity of inputs in the initial population. This distribution ensures that inputs that were not selected so far are more likely to be selected, so that the initial roofer population can be more diverse.

(3) in_1 is added to I.

(4) Adding this input may result in making other inputs in I redundant. Since this only affects the inputs that overlap with in_1 (§ 7.5), there are at most only a few redundant inputs, and the removal steps can be exhaustively computed, replacing I by $reduc(I)$.

After repeating these four steps, if I is not already in $Roofers(0)$, then it is added to it.

8.3 Parents Selection

For each generation n, parents are selected as follows. If $Misers(n) \neq \varnothing$, then one parent is selected from the miser population and one from the roofer population. Otherwise, two distinct parents are selected from the roofer population. A parent $I_1 \in Misers(n)$ is randomly selected from the miser population using the following distribution:

$$\mathrm{P}_{misers}(I_1) \stackrel{\text{def}}{=} \frac{\frac{1}{exposure(I_1)}}{\sum_{I_2 \in Misers(n)} \frac{1}{exposure(I_2)}}$$

where the exposure is defined in § 8.1. Again, the purpose of this distribution is to ensure that input sets with large coverage or at least large potential (§ 3.3) are more likely to be selected. A parent $I_1 \in Roofers(n)$ is randomly selected from the roofer population using the following distribution:

$$\mathrm{P}_{roofers}(I_1) \stackrel{\text{def}}{=} \frac{\frac{1}{cost(I_1)}}{\sum_{I_2 \in Roofers(n)} \frac{1}{cost(I_2)}}$$

The purpose of this distribution is to ensure that less costly input sets are more likely to be selected.

8.4 Parents Crossover

After selecting two distinct parents I_1 and I_2, we detail how they are used to generate the offspring I_3 and I_4. Our crossover strategy exploits the fact that, for each objective bl to cover, it is easy to infer inputs in $Inputs(bl)$ able to cover bl (§ 3.2). For each crossover, we randomly split the objectives in two halves O_1 and O_2 such that $O_1 \cup O_2 = Coverage_{obj}(C)$ and $O_1 \cap O_2 = \varnothing$. We consider here a balanced split, to prevent cases where one parent massively contributes to offspring coverage.

Then, we use this split to define the crossover: inputs in the overlapping component C are split between $S_1 \stackrel{\text{def}}{=} Inputs(O_1)$, the ones covering the first half of the objectives, and $S_2 \stackrel{\text{def}}{=} Inputs(O_2)$, the ones covering the second half. Note that some inputs may cover objectives both in O_1 and O_2, so we call the *edge* of the split the intersection $S_1 \cap S_2$. Because we assume both parents are reduced, this means that redundant inputs can only happen at the edge of the split. This is another reason for our crossover strategy: by using a crossover based on the objective coverage, we ensure that

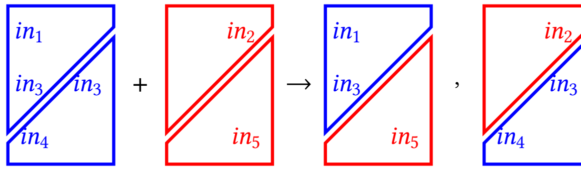


Fig. 5. Crossover from Example 7

only a few redundant inputs are likely to occur after recombination, so that we can reduce them with only a few removal steps. The genetic material of both parents I_1 and I_2 is then split in two parts: inputs in S_1 and inputs in S_2, as follows:

$$I_1 = (I_1 \cap S_1) \cup (I_1 \cap S_2)$$
$$I_2 = (I_2 \cap S_1) \cup (I_2 \cap S_2)$$

Then, these parts are swapped to generated the offspring, as follows:

$$I_3 \overset{\text{def}}{=} (I_1 \cap S_1) \cup (I_2 \cap S_2)$$
$$I_4 \overset{\text{def}}{=} (I_2 \cap S_1) \cup (I_1 \cap S_2)$$

Example 7. For illustration purpose we consider in Figure 5 a small overlapping component $C = \{in_1, in_2, in_3, in_4, in_5\}$ and the following split for the inputs: $Inputs(O_1) = \{in_1, in_2, in_3\}$ and $Inputs(O_2) = \{in_3, in_4, in_5\}$. The edge of the split is $Inputs(O_1) \cap Inputs(O_2) = \{in_3\}$. The offspring of parents $I_1 = \{in_1, in_3, in_4\}$ and $I_2 = \{in_2, in_5\}$ is $I_3 = \{in_1, in_3, in_5\}$ and $I_4 = \{in_2, in_3, in_4\}$.

Note that, according to § 8.4, if both parents are roofers then they cover all the objectives and so do their offspring. Hence, a miser can only appear during the mutation step in § 8.5.

Nevertheless, the swapping of genetic material may lead to some inputs in the offspring being redundant. But, at this stage, MOCCO does not remove them and it waits for the mutation step to occur. The purpose is to benefit from the cumulative effect of the randomness from both crossover and mutation steps, as detailed at the end of § 8.5.

8.5 Offspring Mutation

Each gene of an offspring I corresponds to an input $in \in C$ in the considered overlapping component, the gene value being 1 if $in \in I$ and 0 otherwise. A mutation happens when this gene value is changed, hence mutation adds or removes inputs from an offspring. For each offspring I, we randomly select an input $in \in C$ using a uniform distribution. Then, we replace I by:

$$mutant(I, in) \overset{\text{def}}{=} \begin{cases} I \setminus \{in\} & \text{if } in \in I \\ I \cup \{in\} & \text{otherwise} \end{cases}$$

Since mutation can remove an input, this would result in an empty mutant if the offspring I contained only the removed input. The purpose of the crossover is to explore new combinations of inputs between parents, so an empty mutant would not be beneficial to the search. Hence, in that case, mutation is not performed.

Moreover, an input removal during mutation may reduce cost, in which case the mutation is beneficial, but it may also impact coverage, which would be detrimental. If the removed input was redundant, randomly removing it may lead to a different individual than the one that would be obtained after removal steps, hence exploring a new direction. If the removed input was necessary, then the offspring becomes a miser. In this case, for each uncovered objective, its potential (§ 3.3) is computed. Misers with large potential are likely to be selected as parent (§ 8.3) and the uncovered objectives can then be covered by a subsequent offspring through crossover (with a roofer) or mutation (when an input is added). Therefore, removing an input, even a necessary one, may make the coverage of the impacted objectives more efficient later.

In any case, the crossover (at the edge of the split) and mutation (when an input is added) steps may result in inputs being redundant in the offspring. Since changes in redundancy value could happen only amongst neighbors (for the overlapping relation) of the changed inputs (§ 7.5), we only expect a few redundant inputs. Therefore, removal steps can be exhaustively computed to replace each offspring I by its reduced counterpart $reduc(I)$ (§ 8.2). We waited for the mutation step to occur before removing redundant inputs because, to improve the exploration of the search space, we want the randomness from the crossover (§ 8.4) to interact with the randomness of the mutation. Indeed, performing removal steps before mutation could prevent the search from reaching interesting candidates. For example, the input removed through mutation may differ from the one that would have been selected by removal steps. Further, when the mutation adds an input, it randomly generates more redundancies, potentially leading to a less costly individual after removal steps. In short, premature removal steps could prevent the search from exploring relevant combinations of inputs.

8.6 Population Update

We presented how, starting from the roofer and miser populations, $Roofers(n)$ and $Misers(n)$, at generation n, MOCCO generates, mutates, and then reduces the offspring. We detail here how this offspring is used to obtain the populations $Roofers(n+1)$ and $Misers(n+1)$ at generation $n+1$.

First, we discard any offspring that is a duplicate of an individual already present in either $Roofers(n)$ or $Misers(n)$. Indeed, the duplication of individuals would only result in altering their weight for being selected as parents and thus the purpose of the selection procedure (§ 8.3). Moreover, the roofer population has a fixed size. Hence, for duplicates to be part of the population, other roofers would have to be removed, leading to a reduction of diversity in the population. Finally, as detailed below, each miser is characterized by its fitness vector in order to be compared with other misers. Hence, including duplicates would only make these comparisons more costly without bringing any benefit regarding diversity on the Pareto front.

If a remaining offspring I_1 covers all the objectives in $Coverage_{obj}(C)$, then it is a candidate for the roofer population. Otherwise, it is a candidate for the miser population.

For each candidate I_1 for the roofer population, MOCCO computes its cost. If $cost(I_1) \leq \max \{cost(I) \mid I \in Roofers(n)\}$, then it randomly selects the most costly roofer I_2 (or, in case of a tie, one of the most costly roofers), removes I_2 from the roofer population, and adds I_1 to the population. Otherwise, I_1 is rejected. Note that we chose $\leq$ instead of $<$ for the above cost criterion because, in case of a tie, we prefer to evolve the population instead of maintaining the status quo, to increase the odds of exploring new regions of the search space. After completing this procedure for each candidate to the roofer population, we obtain the population $Roofers(n+1)$ for the next generation. We prove in the appendix (Theorem 4) that roofers satisfy desirable properties, including that the cheapest roofer of the last generation has not only full coverage but is the cheapest roofer observed during the search to date.

We now detail how the miser population is updated. For each candidate I_1 to the miser population, we compute its fitness vector $F_C(I_1)$ (§ 7.6). Then, for each $I_2 \in Misers(n)$ we compare $F_C(I_1)$ and $F_C(I_2)$. If $I_2 \succ I_1$ in the sense of Pareto-dominance (§ 2.3.1), then we stop the process and I_1 is rejected. If $I_1 \succ I_2$, then I_2 is removed from $Misers(n)$. That way, we ensure that the miser population contains only non-dominated individuals. After completing the comparisons, if the process was not stopped, then I_1 itself is non-dominated, so it is added to the miser population. In that case, $F_C(I_1)$ is stored for future comparisons. After completing this procedure for each candidate to the miser population, we obtain the population $Misers(n+1)$ for the next generation. We prove in the appendix (Theorem 5) that misers satisfy desirable properties during the search, including that no miser encountered during the search Pareto-dominates misers in the last generation.

8.7 Termination

MOCCO repeats the process until it reaches a fixed number n_{gens} of generations. Then, amongst the least costly roofers, it randomly selects one individual I_C (the solution may not be unique as several may have the same cost) as solution to our sub-problem (§ 7.6). I_C covers all the objectives and, amongst the input sets covering those objectives, I_C has the smallest cost encountered during the search. Some misers may have a lower cost, but they do not cover all the objectives, and hence cannot be considered a solution to our sub-problem.

9 STEP 6: DATA POST-PROCESSING

The set I_{necess} was initially empty (§ 7.2) and then accumulated necessary inputs each time redundancy was determined (§ 7.3). After removing inputs and reducing the objectives to be covered accordingly (Section 7), IMPRO obtained a set I_{search} of remaining inputs and objectives. Then, IMPRO divided the remaining problem into subproblems (§ 7.6), one for each overlapping component C. Finally, for each overlapping component C, the corresponding subproblem was solved using MOCCO (Section 8), obtaining the corresponding minimized component I_C. At the end of the search, AIM merges inputs from each minimized component I_C with the necessary inputs I_{necess} to obtain a *minimized input set* I_{final} as solution to our initial problem (§ 3.4):

$$I_{final} \stackrel{\text{def}}{=} I_{necess} \cup \bigcup_{C \in OverlapComps(I_{search})} I_C$$

10 EMPIRICAL EVALUATION

In this section, we report our results on the assessment of our approach with two Web systems. We investigate the following Research Questions (RQs):

RQ1 What is the vulnerability detection effectiveness of AIM, compared to alternatives? This research question aims to determine if and to what extent AIM reduces the effectiveness of MST by comparing the vulnerabilities detected between the initial and the minimized input sets. Also, we further compare the vulnerability detection rate of AIM with simpler alternative approaches.

RQ2 What is the input set minimization effectiveness of AIM, compared to alternatives? This research question aims to analyze the magnitude of minimization in terms of the number of inputs, cost (§ 3.1 and 5.1), and execution time for the considered MRs, both for AIM and alternative approaches.

10.1 Experiment Design

10.1.1 Subjects of the Study. To assess our approach with MRs and input sets that successfully detect real-world vulnerabilities, we rely on the same input sets and settings as MST-wi [11].

The targeted Web systems under test are Jenkins [20] and Joomla [30]. Jenkins is a leading open source automation server while Joomla is a content management system (CMS) that relies on the MySQL RDBMS and the Apache HTTP server. We chose these Web systems because of their plug-in architecture and Web interface with advanced features (such as Javascript-based login and AJAX interfaces), which makes Jenkins and Joomla good representatives of modern Web systems.

Further, these systems present differences in their output interface and input types that, since inputs and outputs are key drivers for our approach, contribute to improve the generalizability of our results. Concerning outputs, Joomla is a CMS where Web pages tend to contain a large amount of static text that differ in every page, while Jenkins provides mainly structured content that may continuously change (e.g., seconds from the last execution of a Jenkins task). The input interfaces of Jenkins are mainly short forms and buttons whereas the inputs interfaces of Joomla often include long text areas and several selection interfaces (e.g., for tags annotation).

The selected versions of Jenkins and Joomla (i.e., 2.121.1 and 3.8.7, respectively) are affected by known vulnerabilities that can be triggered from the Web interface; we describe them in § 10.1.2.

The input set provided in the MST-wi's replication package has been collected by running Crawljax with, respectively, four users for Jenkins and six users for Joomla having different roles, e.g., admin. For each role, Crawljax has been executed for a maximum of 300 minutes, to prevent the crawler from running indefinitely, thereby avoiding excessive resource consumption. Further, to exercise features not reached by Crawljax, a few additional Selenium [58]-based test scripts (four for Jenkins and one for Joomla) have been added to the input set. In total, we have 160 initial inputs for Jenkins and 148 for Joomla, which are all associated to a unique identifier.

10.1.2 Security Vulnerabilities. The replication package for MST-wi [15] includes 76 metamorphic relations (MRs). These MRs can identify nine vulnerabilities in Jenkins and three vulnerabilities in Joomla using the initial input set, as detailed in Tables 1 and 2, respectively.

Table 1. Jenkins Vulnerabilities.

CVE	Description	Vuln. Type	Input Identifiers
CVE-2018-1000406 [44]	In the file name parameter of a Job configuration, users with Job / Configure permissions can specify a relative path escaping the base directory. Such path can be used to upload a file on the Jenkins host, resulting in an arbitrary file write vulnerability.	CWE_22	160
CVE-2018-1000409 [45]	A session fixation vulnerability prevents Jenkins from invalidating the existing session and creating a new one when a user signed up for a new user account.	CWE_384	112, 113, 114
CVE-2018-1999003 [48]	Jenkins does not perform a permission check for URLs handling cancellation of queued builds, allowing users with Overall / Read permission to cancel queued builds.	CWE_280, CWE_863	116, 157
CVE-2018-1999004 [49]	Jenkins does not perform a permission check for the URL that initiates agent launches, allowing users with Overall / Read permission to initiate agent launches.	CWE_863, CWE_285	2, 116
CVE-2018-1999006 [50]	A exposure of sensitive information vulnerability allows attackers to determine the date and time when a plugin was last extracted.	CWE_200, CWE_668	33, 55, 57, 61, 62, 63, 64, 75, 107, 108, 110, 135, 136, 156, 160
CVE-2018-1999046 [51]	Users with Overall / Read permission are able to access the URL serving agent logs on the UI due to a lack of permission checks.	CWE_200	2, 116
CVE-2020-2162 [52]	Jenkins does not set Content-Security-Policy headers for files uploaded as file parameters to a build, resulting in a stored XSS vulnerability.	CWE_79	1, 18, 19, 23, 26, 75, 156, 158
Password aging problem in Jenkins	Jenkins does not integrate any mechanism for managing password aging; consequently, users aren't incentivized to update passwords periodically.	CWE_262	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 18, 19, 20, 22, 23, 24, 25, 26, 27, 28, 30, 32, 110, 33, 34, 35, 38, 39, 41, 42, 43, 44, 45, 46, 47, 58, 61, 62, 64, 65, 66, 69, 70, 71, 73, 74, 75, 104, 108, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 133, 134, 143, 145, 146, 159, 160
Weak password in Jenkins	Jenkins does not require users to have strong passwords, which makes it easier for attackers to compromise user accounts.	CWE_521	112, 113, 114

For both tables, the first column contains, when available, the CVE identifiers of the considered vulnerabilities. The password aging problem (for both Jenkins and Joomla) and weak password

Table 2. Joomla Vulnerabilities.

CVE	Description	Vuln. Type	Input Identifiers
CVE-2018-11327 [46]	Inadequate checks allow users to see the names of tags that were either unpublished or published with restricted view permission.	CWE_200	37 with 22, 23, 24, 25, 50
CVE-2018-17857 [47]	Inadequate checks on the tag search fields can lead to an access level violation.	CWE_863	1 with 22, 23, 24, 25
Password aging problem in Joomla	Joomla does not integrate any mechanism for managing password aging; consequently, users aren't incentivized to update passwords periodically.	CWE_262	2, 3, 5, 6, 7, 8, 11, 12, 15, 17, 20, 22, 23, 24, 25, 26, 27, 28, 29, 30, 66, 110, 144, 146

(for Jenkins) are vulnerabilities that were identified during the MST-wi study [11] and therefore do not have CVE identifiers. The second column provides a short description of the vulnerabilities. The third column reports the CWE ID for each vulnerability. We present two CWE IDs (e.g., CWE 863 and 280) in cases where the CVE report denotes a general vulnerability type (e.g., CWE 863 for incorrect authorization [48]), though a more precise identification (e.g., CWE 280 concerning improper handling of privileges that may result in incorrect authorization) could be applied. Since the 12 considered vulnerabilities are associated to nine different CWE IDs and each vulnerability has a unique CWE ID, we can conclude that the selected subjects cover a diverse set of vulnerability types, thus further improving the generalizability of our results.

The last column in Tables 1 and 2 lists identifiers for inputs which were able to trigger the vulnerability using one of the corresponding MRs. For instance, one can detect vulnerability CVE-2018-1999046 in Jenkins by running the MR written for CWE_200 with inputs 2 or 116. For the first two Joomla vulnerabilities, two inputs need to be present at the same time in the input set in order to trigger the vulnerability because, as opposed to most MRs, the corresponding MRs requires two source inputs to generate follow-up inputs. For instance, to detect vulnerability CVE-2018-17857 in Joomla, one needs input 1 and at least one input amongst inputs 22, 23, 24, or 25.

10.1.3 AIM configurations. AIM can be configured in different ways to obtain a minimized input set from an initial input set. Such a *configuration* consists in a choice of distance function and algorithm for output clustering (§ 6.2), and a choice of algorithm for action clustering (§ 6.3).

For the sake of conciseness, in Tables 3 to 9, we denote each configuration by three letters, where L and B respectively denote the Levenshtein and Bag distances, and K, D, and H respectively denote the K-means, DBSCAN, and HDBSCAN clustering algorithms. For instance, BDH denotes that Bag distance and DBSCAN were used for output clustering, and then HDBSCAN for action clustering.

AIM performs Silhouette analysis (§ 6.1) to determine the hyper-parameters required for these clustering algorithms. We considered the same ranges of values for the hyper-parameters in both output clustering (§ 6.2) and action clustering steps (§ 6.3). For K-means, we select the range $[1, 70]$ for the number of clusters k. In the case of DBSCAN, the range for the distance threshold ϵ is $[2, 10]$ for Jenkins and $[3, 15]$ for Joomla. The range is larger for Joomla because Joomla has a larger number of Web pages than Jenkins. Finally, the range for the minimum number of neighbours n is $[1, 5]$ for both systems. For HDBSCAN, the range for the minimum number n of individuals required to form a cluster is $[2, 8]$ for both systems.

We also determine the hyper-parameters for the genetic search (Section 8). Related work on whole test suite generation successfully relies on a population of 80 individuals [24]. Since we reduced the problem (Section 7) before applying MOCCO independently to each overlapping component, which includes fewer inputs than the whole test suite generation [24], we experimented with a lower population size of 20 individuals. Additionally, we set the number of generations for the genetic algorithm to 100, similar to the value considered in previous work [24].

10.1.4 Baselines. We identify the following baselines against which to compare AIM configurations.

A 2016 survey reported that 57% of metamorphic testing (MT) work used Random Testing (RT) to generate source inputs [57] and, in 2021, 84% of the publications related to MT adopted traditional or improved RT methods to generate source inputs [28]. Hence, RT is an obvious baseline against which to compare AIM. Since each AIM run is performed using 18 different configurations (§ 10.1.3), each leading to a different minimized input set, for a fair comparison, we configure RT to select a number of inputs that matches that number in the largest minimized input set produced by the 18 AIM configurations. Precisely, we randomly select that many inputs from the initial input set to obtain an input set for the RT baseline. Finally, we repeat this process for each AIM run to obtain the same number of runs for AIM and the RT baseline.

Moreover, Adaptive Random Testing (ART) was proposed to enhance the performance of RT. It is based on the intuition that inputs close to each other are more likely to have similar failure behaviors than inputs further away from each other. Thus, ART selects inputs widely spread across the input domain, in order to find failure with fewer inputs than RT [10]. ART is similar to our action clustering step (§ 6.3), since it is based on partitioning the input space and generating new inputs in blocks that are not already covered [28]. So, to perform ART, we use AIM to perform action clustering directly on the initial input set instead of output classes. Then, for each cluster, we randomly select one input that covers it. Finally, we gather the selected inputs to obtain an input set for the ART baseline. Again, we repeat this process for each AIM run. Since we considered the K-means, DBSCAN, and HDBSCAN clustering algorithms, there are three variants of this baseline.

In Tables 3 to 9, R denotes the RT baseline while AK, AD, and AH denote the ART baselines, using respectively the K-means, DBSCAN, and HDBSCAN clustering algorithms.

10.1.5 Evaluation Metrics. To reduce the impact of randomness in our experiment, each configuration and baseline was run 50 times on each system, obtaining one minimized input set for each run. Moreover, for the sake of performance analysis, we also recorded the execution time required by AIM to generate minimized input sets.

For RQ1, we consider the vulnerabilities described in Table 1 for Jenkins and in Table 2 for Joomla. For each system, we consider that a vulnerability for that system is detected by an input set if it contains at least one input able to trigger this vulnerability. For the first two Joomla vulnerabilities requiring pairs of inputs, the vulnerability is detected if both inputs are present in the input set. Hence, for each minimized input set, our metric is the *vulnerability detection rate (VDR)*, i.e., the number of vulnerabilities detected by the minimized input set, divided by the number of vulnerabilities detected by the corresponding initial input set. If VDR is 100%, then we say that the minimized input set has full vulnerability coverage. Since we consider 50 runs for each configuration, we say that a configuration or baseline leads to full vulnerability coverage if all the minimized input sets obtained at the end of these runs have full vulnerability coverage. For each system, we reject configurations and baselines which do not lead to full vulnerability coverage, and then compare the remaining ones to answer RQ2.

RQ2 aims at evaluating the effectiveness of AIM in minimizing the input set. Our goal is to identify the AIM configuration generating minimized input sets leading to the minimal execution time for the 76 considered MRs, across the two case studies, and reporting on the execution time saved, compared to executing MST-wi on the full input set. To have a fair comparison between MRs execution time obtained respectively with the initial and minimized input sets, we have to take into account the AIM execution time required to minimize the initial input set. Thus, the input set minimization effectiveness is quantified as the sum of AIM execution time to obtain the selected minimized input set plus MRs execution time with the minimized input set, divided by that of the initial input set. However, since MR execution time is usually large, we cannot collect the time required to execute our 76 MRs on all the input sets generated by all AIM configurations (1800 runs in total, resulting from 18 configurations × 50 repetitions × 2 case study subjects). For this reason, we rely on three additional metrics, that could be inferred without executing MRs, to identify "best" configurations (detailed in the next paragraph) for both systems and the corresponding minimized input sets. Then, we report on the input set minimization effectiveness obtained by such configuration. To make the experiment feasible, amongst the 50 minimized input sets of the best configuration, we select one which has a median cost (§ 3.1 and 5.1), so that is representative of the 50 runs.

To determine the "best" AIM configuration, we consider the size of the generated input set (i.e., the number of inputs in it), its cost (§ 3.1 and 5.1), and the time required by the configuration to generate results. Input set size is a direct measure of effectiveness, while cost is an indirect measure, specific to our approach, that is linearly correlated with MR execution time (§ 5.1). For these three metrics (size, cost, AIM execution time), we compare, for each system, the AIM configurations and baselines leading to full vulnerability coverage. More precisely, for each metric, we denote by M_i the value of the metric obtained for the i^{th} approach (AIM configurations or baseline); the 50 runs of approach i leading to a sample containing 50 data points.

To compare two samples for M_1 and M_2, we perform a Mann-Whitney-Wilcoxon test, which is recommended to assess differences in stochastic order for software engineering data analysis [3]. This is a non-parametric test of the null hypothesis that $\mathrm{P}(M_1 > M_2) = \mathrm{P}(M_1 < M_2)$, i.e., M_1 and M_2 are stochastically equal [64]. Hence, from M_1 and M_2 samples, we obtain the p-value p indicating how likely is the observation of these samples, assuming that M_1 and M_2 are stochastically equal. If $p \leq 0.05$, we consider it is unlikely that M_1 and M_2 are stochastically equal.

To assess practical significance, we also consider a metric for effect size. An equivalent reformulation of the null hypothesis is $\mathrm{P}(M_1 > M_2) + 0.5 \times \mathrm{P}(M_1 = M_2) = 0.5$, which can be estimated by counting in the samples the number of times a value for M_1 is larger than a value for M_2 (ties counting for 0.5), then by dividing by the number of comparisons. That way, we obtain the Vargha and Delaney's A_{12} metric [64] which, for the sake of conciseness, we simply denote A in Tables 4 to 9. A is considered to be a robust metric for representing effect size in the context of non-parametric methods [32]. A ranges from 0 to 1, where $A = 0$ indicates that $\mathrm{P}(M_1 < M_2) = 1$, $A = 0.5$ indicates that $\mathrm{P}(M_1 > M_2) = \mathrm{P}(M_1 < M_2)$, and $A = 1$ indicates that $\mathrm{P}(M_1 > M_2) = 1$.

10.2 Empirical Results

We first describe the system configurations used to obtain our results (§ 10.2.1). To answer RQ1, we report the VDR associated with the obtained minimized input sets (§ 10.2.2). Then, to answer RQ2, we describe the effectiveness of the input set reduction (§ 10.2.3).

10.2.1 System Configurations. We performed all the experiments on a system with the following configurations: a virtual machine installed on professional desktop PCs (Dell G7 7500, RAM 16Gb, Intel(R) Core(TM) i9-10885H CPU @ 2.40GHz) and terminal access to a shared remote server with Intel(R) Xeon(R) Gold 6234 CPU (3.30GHz) and 8 CPU cores.

10.2.2 RQ1 - Detected Vulnerabilities. Results are presented in Table 3. Configurations and baselines that lead to full vulnerability coverage for both systems are in green, in yellow if they lead to full vulnerability coverage for one system, and in red if they never lead to full vulnerability coverage.

First, note that **the choice of distance function for output clustering does not have a significant impact on vulnerability coverage**. Indeed, apart from LDH and BDH, the results using the Levenshtein or Bag distances are fairly similar (e.g., both LKK and BKK discover 450 vulnerabilities in Jenkins) and seem to only depend on the choice of clustering algorithms. This indicates that the order of words in a Web page is not a relevant distinction when performing

Table 3. Coverage of the Jenkins and Joomla vulnerabilities after 50 runs of each configuration and baseline.

Vulnerability Coverage	System under test			
	Jenkins		Joomla	
Configurations or baselines	Number of detected vulnerabilities	VDR	Number of detected vulnerabilities	VDR
LKK	450	100.0%	146	97.3%
LKD	371	82.4%	50	33.3%
LKH	379	84.2%	150	100.0%
LDK	450	100.0%	150	100.0%
LDD	400	88.9%	50	33.3%
LDH	400	88.9%	50	33.3%
LHK	450	100.0%	100	66.7%
LHD	403	89.6%	100	66.7%
LHH	447	99.3%	100	66.7%
BKK	450	100.0%	133	88.7%
BKD	403	89.6%	50	33.3%
BKH	410	91.1%	150	100.0%
BDK	450	100.0%	150	100.0%
BDD	338	75.1%	50	33.3%
BDH	450	100.0%	50	33.3%
BHK	450	100.0%	100	66.7%
BHD	404	89.8%	100	66.7%
BHH	428	95.1%	100	66.7%
R	339	75.3%	74	49.3%
AK	447	99.3%	125	83.3%
AD	77	17.1%	22	14.7%
AH	350	77.8%	68	45.3%

clustering for vulnerability coverage. Considering now LDH and BDH, taking into account the order of words can even be detrimental, since they perform equally poorly for Joomla but they differ for Jenkins, where only BDH leads to full vulnerability coverage.

Second, **the choice of clustering algorithm for action clustering seems to be the main factor determining vulnerability coverage**. Configurations using DBSCAN as algorithm for the action clustering step never lead to full vulnerability coverage for any system. This indicates that this clustering algorithm poorly fits the data in the input space. This is confirmed by the results obtained for the AD baseline, which only uses DBSCAN on the input space and performs the worst (amongst baselines and AIM configurations) regarding vulnerability coverage. After investigation, the minimized input sets acquired for AD are much smaller compared to those obtained for the other baseline methods. These results cannot be explained by the hyper-parameter as we employed a large range of values (§ 10.1.3). We conjecture that DBSCAN merges together many action clusters even when the URLs involved in these actions are distinct.

On the other hand, **configurations using K-means for the action clustering step always lead to full vulnerability coverage for Jenkins and lead to the largest vulnerability coverage for Joomla**. This is confirmed by the results obtained for the AK baseline, which only uses K-means on the input space and performs the best (amongst baselines) regarding vulnerability coverage. Indeed, even if this configuration does not lead to full vulnerability coverage, it is very close. In fact, even if it tends to perform worse than AIM configurations that use K-means for action clustering, it tends to perform better than AIM configurations that do not use K-means for action clustering. The success of K-means in achieving better vulnerability coverage on these datasets can be attributed to its ability to handle well-separated clusters. In our case, these clusters are well-separated because of the distinct URLs occurring in the datasets.

Finally, **no baseline reached full vulnerability coverage**. On top of the already mentioned AK and AD baselines, AH performed similarly to random testing (R), indicating that the effect of the HDBSCAN algorithm for action clustering is neutral. The only AIM configuration that performed worse than random testing is BDD, combining DBSCAN (as mentioned before, the worst clustering

Table 4. Comparison of Jenkins input set sizes for configurations with full vulnerability coverage.

sizes		LKK	**LDK**	LHK	BKK	**BDK**	BDH	BHK
LKK	p		5.4e−15	5.1e−1	4.2e−1	8.8e−1	3.1e−20	7.2e−1
	A		0.05	0.54	0.55	0.49	1.0	0.48
LDK	p	5.4e−15		1.8e−17	1.4e−15	1.8e−14	3.2e−20	3.8e−14
	A	0.95		0.99	0.96	0.94	1.0	0.94
LHK	p	5.1e−1	1.8e−17		9.8e−1	3.4e−1	3.0e−20	1.6e−1
	A	0.46	0.01		0.5	0.44	1.0	0.42
BKK	p	4.2e−1	1.4e−15	9.8e−1		4.1e−1	3.2e−20	2.1e−1
	A	0.45	0.04	0.5		0.45	1.0	0.43
BDK	p	8.8e−1	1.8e−14	3.4e−1	4.1e−1		3.2e−20	6.9e−1
	A	0.51	0.06	0.56	0.55		1.0	0.48
BDH	p	3.1e−20	3.2e−20	3.0e−20	3.2e−20	3.2e−20		3.2e−20
	A	0.0	0.0	0.0	0.0	0.0		0.0
BHK	p	7.2e−1	3.8e−14	1.6e−1	2.1e−1	6.9e−1	3.2e−20	
	A	0.52	0.06	0.58	0.57	0.52	1.0	

Table 5. Comparison of Jenkins input set costs for configurations with full vulnerability coverage.

costs		LKK	**LDK**	LHK	BKK	**BDK**	BDH	BHK
LKK	p		2.5e−16	5.9e−5	1.5e−2	4.4e−3	4.1e−18	5.4e−3
	A		0.02	0.73	0.64	0.67	1.0	0.66
LDK	p	2.5e−16		7.0e−18	9.5e−18	7.0e−18	4.1e−18	7.0e−18
	A	0.98		1.0	1.0	1.0	1.0	1.0
LHK	p	5.9e−5	7.0e−18		1.0e−1	2.4e−1	4.1e−18	1.4e−1
	A	0.27	0.0		0.41	0.43	1.0	0.41
BKK	p	1.5e−2	9.5e−18	1.0e−1		5.8e−1	4.1e−18	6.1e−1
	A	0.36	0.0	0.59		0.53	1.0	0.53
BDK	p	4.4e−3	7.0e−18	2.4e−1	5.8e−1		4.1e−18	8.2e−1
	A	0.33	0.0	0.57	0.47		1.0	0.49
BDH	p	4.1e−18	4.1e−18	4.1e−18	4.1e−18	4.1e−18		4.1e−18
	A	0.0	0.0	0.0	0.0	0.0		0.0
BHK	p	5.4e−3	7.0e−18	1.4e−1	6.1e−1	8.2e−1	4.1e−18	
	A	0.34	0.0	0.59	0.47	0.51	1.0	

algorithm regarding vulnerability detection) for both output and action clustering with Bag distance. Only LDK and BDK lead to full vulnerability coverage for both Jenkins and Joomla, and hence are our candidate "best" configurations in terms of VDR. The combination of DBSCAN and K-means was very effective on our dataset since DBSCAN was able to identify dense regions of outputs and K-means allowed for further refinement, forming well-defined action clusters based on URLs.

10.2.3 RQ2 - Input Set Reduction Effectiveness. To answer RQ2 on the effectiveness of minimization, we compare the input set reduction of baselines and configurations for both Jenkins and Joomla. Amongst them, only the LKK, LDK, LHK, BKK, BDK, BDH, and BHK configurations lead to full vulnerability coverage for Jenkins. Their input set sizes are compared in Table 4, their costs in Table 5, and their AIM execution time in Table 6. Similarly, only the LKH, LDK, BKH, and BDK configurations lead to full vulnerability coverage for Joomla. Their input set sizes are compared in Table 7, their costs in Table 8, and their AIM execution time in Table 9. Configurations with full vulnerability coverage for both Jenkins and Joomla (i.e., LDK and BDK) are in bold.

In these six tables, configurations in each row are compared with configurations in each column. p denotes the statistical significance and A the effect size (§ 10.1.5). When $p > 0.05$, we consider the metric values obtained from the two configurations not to be significantly different, and hence the cell is left white. Otherwise, the cell is colored, either in green or red. Since we consider input set size and cost and AIM execution time, the smaller the values the better. Thus, green (resp. red) indicates that the configuration in the row is better (resp. worse) than the configuration in the column. The intensity of the color is proportional to the effect size. More precisely, the intensity is

Table 6. Comparison of Jenkins AIM execution times for configurations with full vulnerability coverage.

times		LKK	**LDK**	LHK	BKK	**BDK**	BDH	BHK
LKK	p A		1.5e−6 0.78	1.0e−11 0.89	2.0e−2 0.63	3.8e−5 0.74	3.1e−18 1.0	1.3e−9 0.85
LDK	p A	1.5e−6 0.22		1.2e−5 0.75	3.9e−3 0.33	5.4e−1 0.54	2.6e−18 1.0	1.1e−3 0.69
LHK	p A	1.0e−11 0.11	1.2e−5 0.25		1.7e−9 0.15	1.4e−3 0.32	3.2e−17 0.98	8.7e−1 0.49
BKK	p A	2.0e−2 0.37	3.9e−3 0.67	1.7e−9 0.85		8.0e−3 0.65	3.0e−18 1.0	6.7e−7 0.79
BDK	p A	3.8e−5 0.26	5.4e−1 0.46	1.4e−3 0.68	8.0e−3 0.35		1.5e−17 0.99	1.1e−2 0.65
BDH	p A	3.1e−18 0.0	2.6e−18 0.0	3.2e−17 0.02	3.0e−18 0.0	1.5e−17 0.01		6.3e−16 0.04
BHK	p A	1.3e−9 0.15	1.1e−3 0.31	8.7e−1 0.51	6.7e−7 0.21	1.1e−2 0.35	6.3e−16 0.96	

Table 7. Comparison of Joomla input set sizes for configurations with full vulnerability coverage.

sizes		LKH	**LDK**	BKH	**BDK**
LKH	p A		4.8e−15 0.06	1.3e−1 0.42	2.9e−16 0.06
LDK	p A	4.8e−15 0.94		1.1e−14 0.94	4.5e−16 0.94
BKH	p A	1.3e−1 0.58	1.1e−14 0.06		7.4e−16 0.06
BDK	p A	2.9e−16 0.94	4.5e−16 0.06	7.4e−16 0.94	

Table 8. Comparison of Joomla input set costs for configurations with full vulnerability coverage.

costs		LKH	**LDK**	BKH	**BDK**
LKH	p A		1.3e−14 0.06	6.9e−1 0.48	3.5e−15 0.05
LDK	p A	1.3e−14 0.94		1.3e−14 0.94	7.0e−15 0.94
BKH	p A	6.9e−1 0.52	1.3e−14 0.06		3.6e−15 0.05
BDK	p A	3.5e−15 0.95	7.0e−15 0.06	3.6e−15 0.95	

Table 9. Comparison of Joomla AIM execution times for configurations with full vulnerability coverage.

times		LKH	**LDK**	BKH	**BDK**
LKH	p A		2.5e−18 0.0	3.6e−3 0.33	1.2e−18 0.0
LDK	p A	2.5e−18 1.0		2.8e−18 1.0	1.5e−18 1.0
BKH	p A	3.6e−3 0.67	2.8e−18 0.0		1.3e−18 0.0
BDK	p A	1.2e−18 1.0	1.5e−18 0.0	1.3e−18 1.0	

$|\delta|$, where $\delta = 2 \times A - 1$ is Cliff's delta [32]. $|\delta|$ is a number between 0 and 1, where 0 indicates the smallest intensity (the lightest color) and 1 indicates the largest intensity (the darkest color).

For Jenkins, among the candidate best configurations (i.e., LDK and BDK), **BDK performed significantly better than LDK for input set size and cost**, and even if the difference is smaller for AIM execution time, the effect size is also in favor of BDK. As for the other configurations, Table 4 on input set sizes and Table 5 on input set costs consistently indicate that BDH is the best configuration while LDK is the worst configuration. The other configurations seem equivalent in terms of size.

Table 10. Comparison of MRs execution time before and after input set minimization. The percentage of reduction is one minus the ratio between total execution time after minimization and MRs execution time before minimization.

Execution time (minutes)	**Jenkins**	**Joomla**
MRs with initial input set	38,307	20,703
MRs with minimized input set	6119	3675
+ AIM execution time	22	22
= Total execution time	6141	3697
Percentage of Reduction	**84%**	**82%**

Regarding cost, LKK tends to be the second to last configuration, the other configurations being equivalent. Regarding AIM execution time in Table 6, the results are more nuanced, BDH is again the best configuration, but this time LKK is the worst configuration instead of LDK. BDH is the only configuration that reached full vulnerability coverage for Jenkins without using the K-means clustering algorithm and it performs significantly better than the other configurations, especially the ones involving K-means for both output and action clustering steps. This indicates, without surprise, that the K-means algorithm takes more resources to be executed. BDH did not lead to full vulnerability coverage for Joomla, so we do not consider it as a candidate for "best" configuration.

For Joomla, **BDK performed significantly better than LDK** for the considered metrics. As for the other configurations, Table 7 for input set sizes and Table 8 for input set costs provide identical results, indicating that LKH and BKH dominate the others while being equivalent. Moreover, BDK dominates LDK, which is the worst configuration. The results are almost identical for AIM execution time in Table 9, with the small difference that LKH performs slightly better than BKH. However, LKH and BKH did not lead to full vulnerability coverage for Jenkins, as opposed to BDK and LDK.

Since we obtained similar results for both Jenkins and Joomla, we consider BDK to be the "best" AIM configuration. This is not surprising since Bag distance is less costly to compute than Levenshtein distance (§ 6.2) and we already observed that the order of words in a Web page does not appear to be a relevant distinction for vulnerability coverage (§ 10.2.2).

As mentioned in § 10.2.2, no baseline leads to full vulnerability coverage. AD fared poorly and AH performed similarly to random testing R, but AK was much better, with 99.3% VDR for Jenkins and 83.3% for Joomla. But even if AK had reached full vulnerability coverage for both systems, it would be at a disadvantage compared to AIM configurations. Indeed, over 50 Jenkins runs, the average input set size for AK was 94.92 inputs, while it ranges from 38 inputs (40% of AK) for BDH to 74.8 inputs (79%) for LDK. The average input set cost for AK was 193,698.94 actions, while it ranges from 70,500.76 actions (36%) for BDH to 152,373.54 actions (79%) for LDK. Over 50 Joomla runs, the average input set size for AK was 70 inputs, while it ranges from 36.02 inputs (51%) for LKH to 41.46 inputs (59%) for LDK. The average input set cost for AK was 2,312,784.58 actions, while it ranges from 580,705.24 actions (25%) for LKH to 872,352.72 actions (38%) for LDK. In short, **all AIM configurations with full vulnerability coverage outperformed the best baseline AK**, which highlights the relevance of our approach in reducing the cost of testing.

Finally, in Table 10, we present the results of executing the MRs using both the initial input set and the minimized input set derived from the best configuration. In total, by applying AIM, we reduced the execution time of all 76 MRs from 38,307 minutes to 6119 minutes for Jenkins and from 20,703 minutes to 3675 minutes for Joomla. Moreover, executing AIM to obtain this minimized input set required 22 minutes for both systems. Hence, we have a total execution time of 6141 minutes for Jenkins and 3697 minutes for Joomla. As a result, the ratio of the total execution time for the minimized input sets divided by the execution time for the initial input sets is 16.03% for

Jenkins and 17.85% for Joomla. In other words, **AIM reduced the execution time by about 84% for Jenkins and more than 82% for Joomla**. This large reduction in execution time demonstrates the effectiveness of our approach in reducing the cost of metamorphic security testing.

11 THREATS TO VALIDITY

In this section, we discuss internal, conclusion, construct, and external validity according to conventional practices [65].

11.1 Internal Validity

A potential internal threat concerns inadequate data pre-processing, which may adversely impact our results. Indeed, clustering relies on the computed similarity among the pre-processed outputs and inputs. To address this potential concern, we have conducted a manual investigation of the quality of the clusters obtained without pre-processing. This led us to remove, from the textual content extracted from each Web page, all the content that was shared by many Web pages, like system version, date, or (when present) the menu of the Web page.

For RQ1 on vulnerability detection, one potential threat we face is missing inputs that would be able to exercise a vulnerability or incorrectly considering that an input is able to exercise a vulnerability. To ensure our list of inputs triggering vulnerabilities is complete, one author inspected all the MST-wi execution logs to look for failures.

11.2 Conclusion Validity

For RQ2, we rely on a non-parametric test (i.e., Mann-Whitney-Wilcoxon test) to evaluate the statistical and practical significance of differences in results, computing p-value and Vargha and Delaney's A_{12} metric for effect size. Moreover, to deal with the randomness inherent to search algorithms, all the configurations and baselines were executed over 50 runs.

Randomness may also arise from (1) the workload of the machines employed to conduct experiments, potentially slowing down the performance of MST-wi, AIM, and the case study subjects, and (2) the presence of other users interacting with the software under test, which can impact both execution time and system outputs. To address these concerns, we conducted experiments in dedicated environments, ensuring that the study subjects were exclusively utilized by AIM.

11.3 Construct Validity

The constructs considered in our work are vulnerability detection effectiveness and input set reduction effectiveness. Vulnerability detection effectiveness is measured in terms of vulnerability detection rate. Reduction effectiveness is measured in terms of MR execution time, size and cost of the minimized input set, and AIM execution time for each configuration. As it is expensive to execute all 18 configurations on the MRs, we consider the size of the input set and its cost to select the most efficient configuration. The cost of the input set has been defined in section § 5.1 and shown to be linearly correlated with MR execution time, thus enabling us to evaluate the efficiency of the results.

Finally, we executed the minimized input set obtained from the best configuration on the MRs and compared the obtained execution time, plus the AIM execution time required to minimize the initial input set, with the MRs execution time obtained with the initial input set. Execution time is a direct measure, allowing us to evaluate whether, for systems akin to our case study subjects, AIM should be adopted for making vulnerability testing more efficient and scalable.

11.4 External Validity

One threat to the generalizability of our results stems from the benchmark that we used. It includes 160 inputs for Jenkins and 148 inputs for Joomla. Furthermore, we considered the list of vulnerabilities in Jenkins and Joomla that were successfully triggered with MST-wi. However, even if in this study we used MST-wi to collect our data, the AIM approach does not depend on a particular data collector, and using or implementing another data collector would enable the use of our approach with other frameworks. Moreover, even if we relied on previously obtained MRs to be sure they detect vulnerabilities in the considered Web systems, AIM is a general approach for metamorphic security testing which does not depend on the considered MRs. Finally, in § 10.1.1, we highlighted that the different input/output interfaces provided by Jenkins and Joomla, along with the diverse types of vulnerabilities they contain, is in support of the generalizability of our results. Furthermore, the AIM approach can be generalized to other Web systems, if the data collection and pre-processing components are updated accordingly. Nevertheless, further studies involving systems with known vulnerabilities are needed.

12 RELATED WORK

MT enables the execution of a SUT with a potentially infinite set of inputs thus being more effective than testing techniques requiring the manual specification of either test inputs or oracles. However, in MT, the manual definition of metamorphic relations (MRs) is an expensive activity because it requires that engineers first acquire enough information on the subject under test and then analyze the testing problem to identify MRs. For this reason, in the past, researchers focused on both the definition of methodologies supporting the identification of MRs [17, 60] and the development of techniques for the automated generation of MRs, based on meta-heuristic search [7, 66] and natural language processing [13], and targeting query-based systems [56] and Cyber-Physical Systems [7].

However, source inputs also impact the effectiveness and performance of MT; indeed, MRs generate follow-up inputs from source inputs and both are executed by the SUT. Consequently, the research community has recently shown increasing interest towards investigating the impact of source inputs on MT. We summarize the most relevant works in the following paragraphs. Note that all these studies focus on general fault detection, while we focus on metamorphic security testing for Web systems. However, our approach could also be applied to fault detection while the approaches below could also be applied to security testing. We therefore compare these approaches without considering their difference in application. However, we excluded from our survey those approaches that study the effect of source and follow-up inputs on the metamorphic testing of systems that largely differ from ours (i.e., sequence alignment programs [62], system validation [67], and deep neural networks [69]). In the following paragraphs, we group the surveyed works into three categories: input generation techniques, input selection techniques, and feedback-directed metamorphic testing.

Input generation techniques for MT use white-box approaches based on knowledge of the source code (mainly, for statement or branch coverage), while we use a black-box approach based on input and output information (Section 6). For instance, a study [55] leveraged the evolutionary search approach EvoSuite [22] to evolve whole input sets in order to obtain inputs that lead to more branch coverage or to different results on the mutated and non-mutated versions of the source code. Another example study [61] leveraged symbolic execution to collect constraints of program branches covered by execution paths, then solved these constraints to generate the corresponding source inputs. Finally, the execution of the generated inputs on the SUT was prioritized based on their contribution regarding uncovered statements. In this case, both generation and prioritization phases were white-box approaches based on branch coverage. Note that, while our approach on input set minimization could be seen as similar to input prioritization, both studies focused on increasing coverage, while we focused on reducing cost while maintaining full coverage.

Input selection techniques share the same objective of our work (i.e., reducing the number of source inputs while maximizing MT effectiveness). Because of its simplicity, random testing (RT) is a common strategy for input selection in MT [28]. A 2016 survey reported that 57% of MT work employed RT to generate source inputs and 34% selected source inputs from already existing test suite [57]. RT was enhanced with Adaptive Random Testing (ART), a technique which selects source inputs spread across the input domain with the aim of finding failures with fewer number of inputs than RT. ART outperforms RT in the context of MT in terms of fault detection [10], which (as in the following studies) was evaluated using the F-measure, i.e., the number of inputs necessary to reveal the first failure. In the AIM approach, our action clustering step (§ 6.3) bears similarities with ART, since we partition inputs based on action parameters which are relevant for our SUT. But, instead of assuming that close inputs lead to close outputs, we directly used SUT outputs during our output clustering step (§ 6.2), since they are inexpensive to obtain compared to executing MRs. Finally, instead of only counting the number of inputs as in the F-measure (or the size of the input set, in the context of input generation), we considered the cost of each source input as the number of executed actions as surrogate measure, which is tailored to reducing MR execution time in the context of Web systems. Instead of focusing only on distances between source inputs as in ART, another study in input selection [28] also investigated distances with follow-up inputs, which is an improvement since usually there are more follow-up inputs than source inputs. This led to the Metamorphic testing-based adaptive random testing (MT-ART) technique, which performed better than other ART algorithms regarding test effectiveness, test efficiency, and test coverage (considering statement, block, and branch coverage). Unfortunately, in the AIM approach, we could not consider follow-up inputs to drive the input selection, since executing MRs to generate these follow-up inputs is too expensive in our context. Since our approach aims at reducing MR execution time, executing MRs would defeat our purpose.

Finally, while studies on MT usually focus either on the identification of effective MRs or on input generation/selection, a recent study proposed *feedback-directed metamorphic testing* (FDMT) [59] to determine the next test to perform (both in terms of source input and MR), based on previous test results. They proposed adaptive partition testing (APT) to dynamically select source inputs, based on input categories that lead to fault detection, and a diversity-oriented strategy for MR selection (DOMR) to select an MR generating follow-up inputs that are as different as possible from the already obtained ones. While this approach is promising in general, it is not adapted to our case, where we consider a fixed set of MRs, MR selection being considered outside the scope of this paper. Moreover, since we aim to reduce MR execution time, we cannot execute them and use execution information to guide source input or MR selection during testing. Finally, in our problem definition (Section 3), we do not consider source inputs independently from each other, which is why we reduced (Section 7) then minimized (Section 8) the cost of the input set as a whole.

13 CONCLUSION

As demonstrated in our previous work [11], metamorphic testing alleviates the oracle problem for the automated security testing of Web systems. However, metamorphic testing has shown to be a time-consuming approach. Our approach (AIM) aims to reduce the cost of metamorphic security testing by minimizing the initial input set while preserving its capability at exercising vulnerabilities. Our contributions include (1) a clustering-based black box approach that identifies similar inputs based on their security properties, (2) IMPRO, an approach to reduce the search space as much as possible, then divide it into smaller independent parts, (3) MOCCO, a novel genetic algorithm which is able to efficiently select diverse inputs while minimizing their total cost, and (4) a testing framework automatically performing input set minimization.

We considered 18 different configurations for AIM and we evaluated our approach on two open-source Web systems, Jenkins and Joomla, in terms of vulnerability detection rate and magnitude of the input set reduction. Our empirical results show that the best configuration for AIM is BDK: Bag distance, DBSCAN to cluster the outputs, and K-means to cluster the inputs. The results show that our approach can automatically reduce MRs execution time by 84% for Jenkins and 82% for Joomla while preserving full vulnerability detection. Across 50 runs, the BDK configuration consistently detected all vulnerabilities in Jenkins and Joomla. We also compared AIM with four baselines common in security testing. Notably, none of the baselines reached full vulnerability coverage. Among them, AK (ART baseline using K-means) emerged as the closest to achieving full vulnerability coverage. All AIM configurations with full vulnerability coverage outperformed this baseline in terms of minimized input set size and cost, demonstrating the effectiveness of our approach in reducing the cost of metamorphic security testing.

REFERENCES

[1] Paul Ammann and Jeff Offutt. 2016. *Introduction to Software Testing*. Cambridge University Press, Cambridge.

[2] Andrea Arcuri. 2010. It Does Matter How You Normalise the Branch Distance in Search Based Software Testing. In *Third International Conference on Software Testing, Verification and Validation*. IEEE, Paris, France, 205–214. https://doi.org/10.1109/ICST.2010.17

[3] Andrea Arcuri and Lionel Briand. 2014. A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. *Softw. Test. Verif. Reliab.* 24, 3 (may 2014), 219–250. https://doi.org/10.1002/stvr.1486

[4] Chetan Arora, Mehrdad Sabetzadeh, Lionel Briand, and Frank Zimmer. 2017. Automated Extraction and Clustering of Requirements Glossary Terms. *IEEE Transactions on Software Engineering* 43, 10 (2017), 918–945. https://doi.org/10.1109/TSE.2016.2635134

[5] Mohammed Attaoui, Hazem Fahmy, Fabrizio Pastore, and Lionel Briand. 2023. Black-Box Safety Analysis and Retraining of DNNs Based on Feature Extraction and Clustering. *ACM Trans. Softw. Eng. Methodol.* 32, 3, Article 79 (apr 2023), 40 pages. https://doi.org/10.1145/3550271

[6] Mohammed Oualid Attaoui, Hazem Fahmy, Fabrizio Pastore, and Lionel Briand. 2023. DNN Explanation for Safety Analysis: an Empirical Evaluation of Clustering-based Approaches. *arXive* (2023). https://doi.org/arXiv:2301.13506

[7] Jon Ayerdi, Valerio Terragni, Aitor Arrieta, Paolo Tonella, Goiuria Sagardui, and Maite Arratibel. 2021. Generating Metamorphic Relations for Cyber-Physical Systems with Genetic Programming: An Industrial Case Study. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1264–1274. https://doi.org/10.1145/3468264.3473920

[8] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[9] Ilaria Bartolini, Paolo Ciaccia, and Marco Patella. 2002. String matching with metric trees using an approximate distance. In *String Processing and Information Retrieval: 9th International Symposium, SPIRE 2002 Lisbon, Portugal, September 11–13, 2002 Proceedings 9*. Springer, Berlin, Heidelberg, 271–283.

[10] Arlinta Christy Barus, Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, and Heinz W. Schmidt. 2016. The impact of source test case selection on the effectiveness of metamorphic testing. In *Proceedings of the 1st International Workshop on Metamorphic Testing* (Austin, Texas) *(MET '16)*. Association for Computing Machinery, New York, NY, USA, 5–11. https://doi.org/10.1145/2896971.2896977

[11] Nazanin Chaleshtari Bayati, Fabrizio Pastore, Arda Goknil, and Lionel C. Briand. 2023. Metamorphic Testing for Web System Security. *IEEE Transactions on Software Engineering* 49, 6 (2023), 3430–3471. https://doi.org/10.1109/TSE.2023.3256322

[12] Matteo Biagiola, Andrea Stocco, Filippo Ricca, and Paolo Tonella. 2019. Diversity-based web test generation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 142–153. https://doi.org/10.1145/3338906.3338970

[13] Arianna Blasi, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Antonio Carzaniga. 2021. MeMo: Automatically identifying metamorphic relations in Javadoc comments for test automation. *Journal of Systems and Software* 181

(2021), 111041. https://doi.org/10.1016/j.jss.2021.111041
[14] Nazanin Bayati Chaleshtari, Yoann Marquer, Fabrizio Pastore, and Lionel Claude Briand. 2024. Replication package. https://zenodo.org/records/10661329.
[15] Nazanin Bayati Chaleshtari, Fabrizio Pastore, Arda Goknil, and Lionel Claude Briand. 2023. Replication package. https://doi.org/10.5281/zenodo.7702754.
[16] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. *ACM Comput. Surv.* 51, 1, Article 4 (jan 2018), 27 pages. https://doi.org/10.1145/3143561
[17] Tsong Yueh Chen, Pak-Lok Poon, and Xiaoyuan Xie. 2016. METRIC: METamorphic Relation Identification based on the Category-choice framework. *Journal of Systems and Software* 116 (2016), 177–190. https://doi.org/10.1016/j.jss.2015.07.037
[18] Kalyanmoy Deb and Himanshu Jain. 2014. An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints. *IEEE Transactions on Evolutionary Computation* 18, 4 (Aug 2014), 577–601. https://doi.org/10.1109/TEVC.2013.2281535
[19] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (April 2002), 182–197. https://doi.org/10.1109/4235.996017
[20] Eclipse Foundation. 2018. Jenkins CI/CD server. https://jenkins.io/.
[21] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD'96)*. AAAI Press, Portland, Oregon, 226–231.
[22] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 416–419. https://doi.org/10.1145/2025113.2025179
[23] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (Feb 2013), 276–291. https://doi.org/10.1109/TSE.2012.14
[24] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Trans. Softw. Eng.* 39, 2 (feb 2013), 276–291. https://doi.org/10.1109/TSE.2012.14
[25] Marshall Jr. Hall. 1959. *The Theory of Groups*. MacMillan, USA.
[26] Hadi Hemmati, Andrea Arcuri, and Lionel Briand. 2013. Achieving scalable model-based testing through test case diversity. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22, 1 (2013), 1–42.
[27] W.E. Howden. 1978. Theoretical and Empirical Studies of Program Testing. *IEEE Transactions on Software Engineering* SE-4, 4 (July 1978), 293–298. https://doi.org/10.1109/TSE.1978.231514
[28] Zhan-wei Hui, Xiaojuan Wang, Song Huang, and Sen Yang. 2021. MT-ART: A Test Case Generation Method Based on Adaptive Random Testing and Metamorphic Relation. *IEEE Transactions on Reliability* 70, 4 (Dec 2021), 1397–1421. https://doi.org/10.1109/TR.2021.3106389
[29] Himanshu Jain and Kalyanmoy Deb. 2014. An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point Based Nondominated Sorting Approach, Part II: Handling Constraints and Extending to an Adaptive Approach. *IEEE Transactions on Evolutionary Computation* 18, 4 (Aug 2014), 602–622. https://doi.org/10.1109/TEVC.2013.2281534
[30] Joomla. 2018. Joomla, https://www.joomla.org/.
[31] Mifa Kim, Tomoyuki Hiroyasu, Mitsunori Miki, and Shinya Watanabe. 2004. SPEA2+: Improving the Performance of the Strength Pareto Evolutionary Algorithm 2. In *Parallel Problem Solving from Nature - PPSN VIII*, Xin Yao, Edmund K. Burke, José A. Lozano, Jim Smith, Juan Julián Merelo-Guervós, John A. Bullinaria, Jonathan E. Rowe, Peter Tiňo, Ata Kabán, and Hans-Paul Schwefel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 742–751.
[32] Barbara Kitchenham, Lech Madeyski, David Budgen, Jacky Keung, Pearl Brereton, Stuart Charters, Shirley Gibbs, and Amnart Pohthong. 2017. Robust Statistical Methods for Empirical Software Engineering. *Empirical Softw. Engg.* 22, 2 (apr 2017), 579–630. https://doi.org/10.1007/s10664-016-9437-5
[33] Bernhard Korte and Rainer Schrader. 1981. On the Existence of Fast Approximation Schemes. In *Nonlinear Programming 4*, Olvi L. Mangasarian, Robert R. Meyer, and Stephen M. Robinson (Eds.). Academic Press, Madison, Wisconsin, 415–437. https://doi.org/10.1016/B978-0-12-468662-5.50020-3
[34] Bingdong Li, Jinlong Li, Ke Tang, and Xin Yao. 2015. Many-Objective Evolutionary Algorithms: A Survey. *ACM Comput. Surv.* 48, 1, Article 13 (sep 2015), 35 pages. https://doi.org/10.1145/2792984
[35] Zheng Li, Mark Harman, and Robert M. Hierons. 2007. Search Algorithms for Regression Test Case Prioritization. *IEEE Transactions on Software Engineering* 33, 4 (April 2007), 225–237. https://doi.org/10.1109/TSE.2007.38
[36] Phu X. Mai, Arda Goknil, Lwin Khin Shar, Fabrizio Pastore, Lionel C. Briand, and Shaban Shaame. 2018. Modeling Security and Privacy Requirements: a Use Case-Driven Approach. *Information and Software Technology* 100 (2018), 165–182. https://doi.org/10.1016/j.infsof.2018.04.007

[37] Phu Xuan Mai, Fabrizio Pastore, Arda Goknil, and Lionel Claude Briand. 2018. A Natural Language Programming Approach for Requirements-Based Security Testing. *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)* (2018), 58–69. https://api.semanticscholar.org/CorpusID:53711718

[38] Phu Xuan Mai, Fabrizio Pastore, Arda Goknil, and Lionel Claude Briand. 2019. Metamorphic Security Testing for Web Systems. *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)* (2019), 186–197. https://api.semanticscholar.org/CorpusID:209202564

[39] Leland McInnes, John Healy, and Steve Astels. 2017. hdbscan: Hierarchical density based clustering. *Journal of Open Source Software* 2, 11 (2017), 205. https://doi.org/10.21105/joss.00205

[40] Sérgio Mergen. 2022. Extending the Bag Distance for String Similarity Search. *SN Comput. Sci.* 4, 2 (dec 2022), 15 pages. https://doi.org/10.1007/s42979-022-01502-5

[41] MITRE. [n. d.]. *CWE-286: Incorrect User Management.* MITRE. https://cwe.mitre.org/data/definitions/286.html

[42] MITRE. [n. d.]. CWE-863: Incorrect Authorization. https://cwe.mitre.org/data/definitions/863.html.

[43] MITRE. [n. d.]. *CWE VIEW: Architectural Concepts.* MITRE. https://cwe.mitre.org/data/definitions/1008.html

[44] MITRE. 2018. *CVE-2018-1000406, concerns CWE-276.* MITRE. Retrieved 2018-11-22 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1000406

[45] MITRE. 2018. *CVE-2018-1000409, concerns OTG-SESS-003.* MITRE. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1999003

[46] MITRE. 2018. *CVE-2018-11327, concerns CWE-200.* MITRE. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-11327

[47] MITRE. 2018. *CVE-2018-17857, concerns CWE-200.* MITRE. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-17857

[48] MITRE. 2018. *CVE-2018-1999003, concerns OTG-AUTHZ-002.* MITRE. Retrieved 2018-11-22 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1999003

[49] MITRE. 2018. *CVE-2018-1999004, concerns OTG-AUTHZ-002.* MITRE. Retrieved 2018-11-22 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1999004

[50] MITRE. 2018. *CVE-2018-1999006, concerns CWE-138.* MITRE. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1999006

[51] MITRE. 2018. *CVE-2018-1999046, concerns OTG-AUTHZ-002.* MITRE. Retrieved 2018-11-22 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1999046

[52] MITRE. 2020. *CVE-2020-2162, concerns OTG-INPVAL-003.* MITRE. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-2162

[53] OWASP. 2023. *Open Web Application Security Project.* OWASP Foundation. https://www.owasp.org/

[54] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2015. Reformulating Branch Coverage as a Many-Objective Optimization Problem. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST).* IEEE, Graz, Austria, 1–10. https://doi.org/10.1109/ICST.2015.7102604

[55] Prashanta Saha and Upulee Kanewala. 2018. Fault detection effectiveness of source test case generation strategies for metamorphic testing. In *Proceedings of the 3rd International Workshop on Metamorphic Testing* (Gothenburg, Sweden) *(MET '18).* Association for Computing Machinery, New York, NY, USA, 2–9. https://doi.org/10.1145/3193977.3193982

[56] Sergio Segura, Juan C. Alonso, Alberto Martin-Lopez, Amador Durán, Javier Troya, and Antonio Ruiz-Cortés. 2023. Automated Generation of Metamorphic Relations for Query-Based Systems. In *Proceedings of the 7th International Workshop on Metamorphic Testing* (Pittsburgh, Pennsylvania) *(MET '22).* Association for Computing Machinery, New York, NY, USA, 48–55. https://doi.org/10.1145/3524846.3527338

[57] Sergio Segura, Gordon Fraser, Ana B. Sanchez, and Antonio Ruiz-Cortés. 2016. A Survey on Metamorphic Testing. *IEEE Transactions on Software Engineering* 42, 9 (Sep. 2016), 805–824. https://doi.org/10.1109/TSE.2016.2532875

[58] Selenium 2018. *Selenium Web Testing Framework, https://www.seleniumhq.org/.* Selenium.

[59] Chang-Ai Sun, Hepeng Dai, Huai Liu, and Tsong Yueh Chen. 2023. Feedback-Directed Metamorphic Testing. *ACM Trans. Softw. Eng. Methodol.* 32, 1, Article 20 (feb 2023), 34 pages. https://doi.org/10.1145/3533314

[60] Chang-Ai Sun, An Fu, Pak-Lok Poon, Xiaoyuan Xie, Huai Liu, and Tsong Yueh Chen. 2021. METRIC^{+}+: A Metamorphic Relation Identification Technique Based on Input Plus Output Domains. *IEEE Transactions on Software Engineering* 47, 9 (2021), 1764–1785. https://doi.org/10.1109/TSE.2019.2934848

[61] Chang-ai Sun, Baoli Liu, An Fu, Yiqiang Liu, and Huai Liu. 2022. Path-directed source test case generation and prioritization in metamorphic testing. *Journal of Systems and Software* 183 (2022), 111091. https://doi.org/10.1016/j.jss.2021.111091

[62] Joshua Y.S. Tang, Andrian Yang, Tsong Yueh Chen, and Joshua W.K. Ho. 2017. Harnessing Multiple Source Test Cases in Metamorphic Testing: A Case Study in Bioinformatics. In *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET).* IEEE, Buenos Aires, Argentina, 10–13. https://doi.org/10.1109/MET.2017.4

[63] Julian Thomé, Lwin Khin Shar, Domenico Bianculli, and Lionel Briand. 2017. Search-driven string constraint solving for vulnerability detection. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, Buenos Aires, Argentina, 198–208.

[64] András Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132. https://doi.org/10.3102/10769986025002101 arXiv:https://doi.org/10.3102/10769986025002101

[65] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. 2012. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, Heidelberg, Germany.

[66] Bo Zhang, Hongyu Zhang, Junjie Chen, Dan Hao, and Pablo Moscato. 2019. Automatic Discovery and Cleansing of Numerical Metamorphic Relations. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Cleveland, USA, 235–245. https://doi.org/10.1109/ICSME.2019.00035

[67] Miao Zhang, Jacky Wai Keung, Tsong Yueh Chen, and Yan Xiao. 2021. Validating class integration test order generation systems with Metamorphic Testing. *Information and Software Technology* 132 (2021), 106507. https://doi.org/10.1016/j.infsof.2020.106507

[68] Shengnan Zhang, Yan Hu, and Guangrong Bian. 2017. Research on string similarity algorithm based on Levenshtein Distance. In *2017 IEEE 2nd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*. IEEE, Chongqing, China, 2247–2251. https://doi.org/10.1109/IAEAC.2017.8054419

[69] Jinyi Zhou, Kun Qiu, Zheng Zheng, Tsong Yueh Chen, and Pak-Lok Poon. 2020. Using Metamorphic Testing to Evaluate DNN Coverage Criteria. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, Coimbra, Portugal, 147–148. https://doi.org/10.1109/ISSREW51248.2020.00055

[70] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. 2001. *SPEA2: Improving the Strength Pareto Evolutionary Algorithm*. Technical Report 103. Computer Engineering and Communication Networks Lab (TIK), Swiss Federal Institute of Technology (ETH), Zurich.

A FORMAL RESULTS FOR THE PROBLEM DEFINITION

We prove in this appendix Theorem 1 presented in Section 3 as well as Lemma 1 and Lemma 2 which are necessary for proofs in Appendix B.1 and Lemma 4 and Lemma 7 which are necessary for proofs in Appendix B.2.

A.1 Input Coverage and Cost

Lemma 1 (Non-Empty Coverage). *For every input in, we have* $Coverage(in) \neq \varnothing$.

Proof. We consider only inputs containing at least one action (§ 6.2). Let $act_1 = action(in, 1)$ be the first action of *in*. act_1 has an output in $outCl_1 = OutputClass(in, 1)$ (§ 6.2.2). act_1 is in the action set $actSet_1 = ActionSet(outCl_1)$ (§ 6.3.1). After clustering of this action set, let $bl_1 = Subclass(in, 1) = ActionSubclass(act_1, actSet_1)$ be the input block of the action act_1 executed in *in* (§ 6.3.6). Therefore, we have $bl_1 \in Coverage(in)$. □

Lemma 2 (Cost is Positive). *For every input in, we have* $cost(in) > 0$. *For every input set I, we have* $cost(I) \geq 0$, *and* $cost(I) = 0$ *if and only if* $I = \varnothing$.

Proof. Since we removed inputs with no cost (§ 5.1), for each input *in*, $cost(in) > 0$. The cost of an input set is the sum of the cost of its inputs, hence $cost(I) \geq 0$. In particular, if $I = \varnothing$, then $cost(I) = 0$. Finally, because the cost is positive, the only case where $cost(I) = 0$ is when $I = \varnothing$. □

A.2 Redundancy

In this section, we prove that our characterization of redundancy is sound regarding the coverage of an input set (Proposition 1), then we introduce Lemma 4 which is useful to prove Lemma 6 in Appendix A.3 and Proposition 8 in Appendix B.2.

First, if an input *in* is in the considered input set *I*, then its redundancy is not negative. Indeed, there is always at least one input in *I* that covers the input blocks covered by *in*, which is *in* itself.

Lemma 3 (Redundancy is Non-negative). *Let I be an input set and in be an input. If* $in \in I$, *then* $redundancy(in, I) \geq 0$.

Proof. Let *in* be an input in *I*. Let $bl \in Coverage(in)$ be any block covered by *in*. Because $bl \in Coverage(in)$, we have that $in \in Inputs(bl)$ (§ 3.1). Moreover, $in \in I$, so $in \in Inputs(bl) \cap I$, thus we have $superpos(bl, I) \geq 1$ (§ 3.3). Therefore, for any $bl \in Coverage(in)$ we have $superpos(bl, I) \geq 1$. So, $\min\{superpos(bl, I) \mid bl \in Coverage(in)\} \geq 1$. By subtracting 1, we have $redundancy(in, I) \geq 0$ (§ 3.3). □

We prove that our characterization of redundancy is sound regarding the coverage of an input set. In other words, if an input is redundant in an input set, then it can be removed without reducing the coverage of the input set:

Proposition 1 (Redundancy Soundness). *Let* $I \subseteq I_{init}$ *be an input set and* $in \in I$. *If* $in \in Redundant(I)$, *then* $Coverage(I \setminus \{in\}) = Coverage(I)$.

Proof. $Coverage(I) = Coverage(I \setminus \{in\}) \cup Coverage(in)$ (§ 6.3.6). We assume that *in* is redundant in *I*. So, according to Lemma 3, we have $redundancy(in, I) > 0$ (§ 3.3). Thus, for each $bl \in Coverage(in)$, we have that $superpos(bl, I) \geq 2$. Hence, according to our definition of superposition (§ 3.3), there are at least two inputs in_1 and in_2 in *I* which also belong to $Inputs(bl)$. Because we have $bl \in Coverage(in)$, we have that $in \in Inputs(bl)$ (§ 3.1). So, *in* is one of these inputs. We assume this is the first one i.e., $in_1 = in$. Therefore, for each $bl \in Coverage(in)$, there exists an input $in_2 \in Inputs(bl) \cap I$ which is not *in*. We denote $in_2 = in_{bl}$ this input. For each $bl \in Coverage(in)$, we have $in_{bl} \in Inputs(bl)$. So, we have $bl \in Coverage(in_{bl})$ (§ 3.1). Moreover,

$in_{bl} \in I$ and is not in, so is in $I \setminus \{in\}$. Thus (§ 6.3.6) we have $bl \in Coverage(I \setminus \{in\})$. Therefore, $Coverage(in) \subseteq Coverage(I \setminus \{in\})$. Finally, $Coverage(I) = Coverage(I \setminus \{in\}) \cup Coverage(in) = Coverage(I \setminus \{in\})$. □

Finally, removing an input in an input set may update the redundancy of other inputs:

Lemma 4 (Removing an Input)**.** *Let I be an input set and $in_2, in_1 \in I$ be two inputs in this input set.*

$$redundancy(in_1, I) - 1 \leq redundancy(in_1, I \setminus \{in_2\}) \leq redundancy(in_1, I)$$

Proof. Let $bl \in Coverage(in_1)$. We denote $c_1 = \mathrm{card}(Inputs(bl) \cap I)$ and $c_2 = \mathrm{card}(Inputs(bl) \cap (I \setminus \{in_2\}))$. If $in_2 \in Inputs(bl)$ then $c_2 = c_1 - 1$. Otherwise, $c_2 = c_1$. Therefore, for every $bl \in Coverage(in_1)$ we have $c_1 - 1 \leq c_2 \leq c_1$. This includes the input blocks minimizing the cardinality in the definition of redundancy (§ 3.3). Hence the result. □

A.3 Valid Orders of Removal Steps

In this section, we first prove Lemma 5 that establishes that orders of removal steps contain inputs without repetition. Then, we introduce the sublists and prove Lemma 7, used in the rest of the section and in proofs of Appendix B.2. Finally, we prove Theorem 1 presented in § 3.3.

Lemma 5 (Order without Repetition)**.** *Let I be an input set. If $[in_1, \dots, in_n] \in ValidOrders(I)$, then $in_1, \dots, in_n$ are distinct.*

Proof. The proof is done by induction on n.

If $n = 0$, then $[in_1, \dots, in_n] = [\,]$ is empty, hence there are no two identical inputs.

We now consider $[in_1, \dots, in_n, in_{n+1}] \in ValidOrders(I)$. By induction hypothesis, $in_1, \dots, in_n$ are distinct. According to the definition of valid order of removal steps (§ 3.3), we have $[in_1, \dots, in_n, in_{n+1}]$ only if $in_{n+1} \in Redundant(I \setminus \{in_1, \dots, in_n\})$.

Since $Redundant(I) = \{in \in I \mid redundancy(in, I) > 0\}$, we have $in_{n+1} \in I \setminus \{in_1, \dots, in_n\}$.

Therefore, in_{n+1} is distinct from the previous inputs $in_1, \dots, in_n$. Hence the result, which concludes the induction step. □

Lemma 6 (Redundant Inputs After Reduction)**.** *Let I be an input set. For each subset of inputs $\{in_1, \dots, in_n\} \subseteq I$, we have $Redundant(I \setminus \{in_1, \dots, in_n\}) \subseteq Redundant(I)$.*

Proof. The proof is done by induction on n.

If $n = 0$ then $I \setminus \{in_1, \dots, in_n\} = I$, hence the result.

Otherwise, we assume by induction that $Redundant(I \setminus \{in_1, \dots, in_n\}) \subseteq Redundant(I)$.

According to Lemma 4, redundancies can only decrease when performing a removal step. So, for every input $in_0 \in Redundant(I \setminus \{in_1, \dots, in_n, in_{n+1}\})$, we have:

$$0 < redundancy(in_0, I \setminus \{in_1, \dots, in_n, in_{n+1}\}) \leq redundancy(in_0, I \setminus \{in_1, \dots, in_n\})$$

So, $Redundant(I \setminus \{in_1, \dots, in_n, in_{n+1}\}) \subseteq Redundant(I \setminus \{in_1, \dots, in_n\}) \subseteq Redundant(I)$. □

Since orders of removal steps contain inputs without repetition (Lemma 5), the index $index(in, \ell)$ of an input in in a removal order ℓ containing in is well defined. We leverage this to define sublists.

Definition 1 (Sublists). Let $[in_1, \dots, in_n]$ and $[in'_1, \dots, in'_m]$ be two orders of removal steps. We say that $[in'_1, \dots, in'_m]$ is a *sublist* of $[in_1, \dots, in_n]$ if $\{in \in [in'_1, \dots, in'_m]\} \subseteq \{in \in [in_1, \dots, in_n]\}$ and, for any two inputs $in_i, in_j \in [in'_1, \dots, in'_m]$, if $index(in_i, [in_1, \dots, in_n]) \leq index(in_j, [in_1, \dots, in_n])$, then $index(in_i, [in'_1, \dots, in'_m]) \leq index(in_j, [in'_1, \dots, in'_m])$, where $index(in, \ell)$ is the index of input in in ℓ.

Lemma 7. *Let I be an input set and $[in_1, \ldots, in_n]$, $[in'_1, \ldots, in'_m]$ be two orders of removal steps in I. If $[in_1, \ldots, in_n] \in ValidOrders(I)$ and $[in'_1, \ldots, in'_m]$ is a sublist of $[in_1, \ldots, in_n]$, then $[in'_1, \ldots, in'_m] \in ValidOrders(I)$.*

Proof. The proof is done by induction on m.

If $m = 0$ then $[in'_1, \ldots, in'_m] = [\,] \in ValidOrders(I)$.

Otherwise, we consider $[in'_1, \ldots, in'_m, in'_{m+1}]$ and we assume by induction that $[in'_1, \ldots, in'_m] \in ValidOrders(I)$.

Because $[in'_1, \ldots, in'_m, in'_{m+1}]$ is a sublist of $[in_1, \ldots, in_n]$, there exists an index i such that $in_i = in'_{m+1}$. By definition of valid removal steps (§ 3.3), we have $in_i \in Redundant(I \setminus \{in_1, \ldots, in_{i-1}\})$.

Moreover, $[in'_1, \ldots, in'_m, in'_{m+1}]$ is a sublist of $[in_1, \ldots, in_i]$, so we have $\{in \in [in'_1, \ldots, in'_m]\} \subseteq \{in \in [in_1, \ldots, in_{i-1}]\}$. Hence, according to Lemma 6 we have $Redundant(I \setminus \{in_1, \ldots, in_{i-1}\}) \subseteq Redundant(I \setminus \{in'_1, \ldots, in'_m\})$. Therefore, $in'_{m+1} = in_i \in Redundant(I \setminus \{in'_1, \ldots, in'_m\})$.

$[in'_1, \ldots, in'_m] \in ValidOrders(I)$ and $in'_{m+1} \in Redundant(I \setminus \{in'_1, \ldots, in'_m\})$ so, according to our definition for valid orders of removal steps (§ 3.3), we have $[in'_1, \ldots, in'_m, in'_{m+1}] \in ValidOrders(I)$. □

Finally, we prove in Proposition 2 that inputs in a valid order of removal steps can be rearranged in any different order and the rearranged order of removal steps is also valid.

Lemma 8 (Transposition of a Valid Order)**.** *Let I be an input set.*

$$\text{If } [in_1, \ldots, in_i, \ldots, in_j, \ldots, in_n] \in ValidOrders(I) \text{ and } 1 \leq i < j \leq n,$$
$$\text{then } [in_1, \ldots, in_j, \ldots, in_i, \ldots, in_n] \in ValidOrders(I)$$

where in_i and in_j were exchanged and the other inputs are left unchanged.

Proof. The proof is done in four steps.

1) $[in_1, \ldots, in_{i-1}, in_j]$ is a sublist of $[in_1, \ldots, in_{i-1}, in_i, \ldots, in_j, \ldots, in_n] \in ValidOrders(I)$. So, according to Lemma 7, we have $[in_1, \ldots, in_{i-1}, in_j] \in ValidOrders(I)$. Note that the first step even applies to the case $i = 1$.

2) If $j = i+1$, then one can go directly to the third step. Otherwise, for each $i < k < j$, we denote:

$$I_{i+1} \stackrel{\text{def}}{=} I \setminus \{in_1, \ldots, in_{i-1}\}$$
$$I_{k+1} \stackrel{\text{def}}{=} I_k \setminus \{in_k\}$$

and, for the sake of conciseness, $I^i_k = I_k \setminus \{in_i\}$ and $I^j_k = I_k \setminus \{in_j\}$.

We now prove by induction on $i \leq k \leq j-1$ that:

$$[in_1, \ldots, in_{i-1}, in_j, in_{i+1}, \ldots, in_k] \in ValidOrders(I)$$

For the initialization $k = i$, we have from the first step:

$$[in_1, \ldots, in_{i-1}, in_j] \in ValidOrders(I)$$

We now assume by induction hypothesis on $i \leq k < j-1$ that:

$$[in_1, \ldots, in_{i-1}, in_j, in_{i+1}, \ldots, in_k] \in ValidOrders(I)$$

We first prove, for each $bl \in Coverage(in_{k+1})$, that:

$$\text{card}(Inputs(bl) \cap I^j_{k+1}) > 1$$

According to Lemma 5, $in_1, \ldots, in_i, \ldots, in_j, \ldots, in_n$ are distinct. We consider two cases.

a) We assume $bl \in Coverage(in_j)$.

Because $I_j^i = I \setminus \{in_1, \ldots, in_{i-1}, in_i, in_{i+1}, \ldots, in_{j-1}\}$ and $I_{k+1}^j = I \setminus \{in_1, \ldots, in_{i-1}, in_{i+1}, \ldots, in_k, in_j\}$, we have $I_j^i \cup \{in_{k+1}\} \subseteq I_{k+1}^j \cup \{in_j\}$. Thus, for each $bl \in Coverage(in_j) \cap Coverage(in_{k+1})$:

$$\mathrm{card}(Inputs(bl) \cap I_j^i) + 1 \leq \mathrm{card}(Inputs(bl) \cap I_{k+1}^j) + 1$$

Because $[in_1, \ldots, in_i, \ldots, in_j, \ldots, in_n] \in ValidOrders(I)$, we have:

$$redundancy(in_j, I_j^i) > 0$$

So, for each $bl \in Coverage(in_j)$, we have:

$$\mathrm{card}(Inputs(bl) \cap I_j^i) > 1$$

Thus, for each $bl \in Coverage(in_j) \cap Coverage(in_{k+1})$, we have:

$$\mathrm{card}(Inputs(bl) \cap I_{k+1}^j) > 1$$

b) We assume $bl \notin Coverage(in_j)$.

For each $bl \in Coverage(in_{k+1}) \setminus \big(Coverage(in_i) \cup Coverage(in_j)\big)$, we have:

$$\mathrm{card}(Inputs(bl) \cap I_{k+1}^i) = \mathrm{card}(Inputs(bl) \cap I_{k+1}^j)$$

Moreover, for each $bl \in Coverage(in_i) \setminus Coverage(in_j)$, we have:

$$\mathrm{card}(Inputs(bl) \cap I_{k+1}^i) + 1 = \mathrm{card}(Inputs(bl) \cap I_{k+1}^j)$$

So, for each $bl \in Coverage(in_{k+1}) \setminus Coverage(in_j)$, we have:

$$\mathrm{card}(Inputs(bl) \cap I_{k+1}^i) \leq \mathrm{card}(Inputs(bl) \cap I_{k+1}^j)$$

Because $[in_1, \ldots, in_i, \ldots, in_j, \ldots, in_n] \in ValidOrders(I)$, we have:

$$redundancy(in_{k+1}, I_{k+1}^i) > 0$$

So, for each $bl \in Coverage(in_{k+1})$, we have:

$$\mathrm{card}(Inputs(bl) \cap I_{k+1}^i) > 1$$

Thus, for each $bl \in Coverage(in_{k+1}) \setminus Coverage(in_j)$, we have:

$$\mathrm{card}(Inputs(bl) \cap I_{k+1}^j) > 1$$

So, we proved by case, for each $bl \in Coverage(in_{k+1})$, that:

$$\mathrm{card}(Inputs(bl) \cap I_{k+1}^j) > 1$$

Thus, we have $redundancy(in_{k+1}, I_{k+1}^j) > 0$.

Moreover, according to Lemma 5, $in_1, \ldots, in_i, \ldots, in_j, \ldots, in_n$ are distinct, so $in_{k+1} \in I_{k+1}^j$. Hence, $in_{k+1} \in Redundant(I_{k+1}^j)$. By induction hypothesis, we have:

$$[in_1, \ldots, in_{i-1}, in_j, in_{i+1}, \ldots, in_k] \in ValidOrders(I)$$

So, according to § 3.3:

$$[in_1, \ldots, in_{i-1}, in_j, in_{i+1}, \ldots, in_k, in_{k+1}] \in ValidOrders(I)$$

which concludes the induction step.

Therefore, we proved by induction:

$$[in_1, \ldots, in_{i-1}, in_j, in_{i+1}, \ldots, in_{j-1}] \in ValidOrders(I)$$

3) We now prove that, for each $bl \in Coverage(in_i)$, we have:

$$\mathrm{card}(Inputs(bl) \cap I_j^j) > 1$$

where I_j^i and I_j^j are defined in step two. We consider two cases.

a) We assume $bl \in Coverage(in_j)$. Because $I_j^i \cup \{in_i\} = I_j^j \cup \{in_j\}$, for each $bl \in Coverage(in_i) \cap Coverage(in_j)$, we have:

$$\text{card}(Inputs(bl) \cap I_j^i) + 1 = \text{card}(Inputs(bl) \cap I_j^j) + 1$$

Because $[in_1, \ldots, in_i, \ldots, in_j, \ldots, in_n] \in ValidOrders(I)$, we have:

$$redundancy(in_j, I_j^i) > 0$$

So, for each $bl \in Coverage(in_j)$, we have:

$$\text{card}(Inputs(bl) \cap I_j^i) > 1$$

Thus, for each $bl \in Coverage(in_i) \cap Coverage(in_j)$, we have:

$$\text{card}(Inputs(bl) \cap I_j^j) > 1$$

b) We assume $bl \notin Coverage(in_j)$. In that case, for each $bl \in Coverage(in_i) \setminus Coverage(in_j)$, we have:

$$\text{card}(Inputs(bl) \cap I_j^i) + 1 = \text{card}(Inputs(bl) \cap I_j^j)$$

We consider two cases.

i) There exists $i + 1 \leq k \leq j - 1$ such that $bl \in Coverage(in_k)$. In that case, let $k_{\max}$ be the largest. So, we have:

$$\left(Inputs(bl) \cap I_j^i\right) \cup \{in_{k_{\max}}\} = Inputs(bl) \cap I_{k_{\max}}^i$$

Hence:

$$\text{card}(Inputs(bl) \cap I_j^i) + 1 = \text{card}(Inputs(bl) \cap I_{k_{\max}}^i)$$

Thus:

$$\text{card}(Inputs(bl) \cap I_j^j) = \text{card}(Inputs(bl) \cap I_{k_{\max}}^i)$$

Because $[in_1, \ldots, in_i, \ldots, in_j, \ldots, in_n] \in ValidOrders(I)$, we have:

$$redundancy(in_{k_{\max}}, I_{k_{\max}}^i) > 0$$

So, because $bl \in Coverage(in_{k_{\max}})$, we have:

$$\text{card}(Inputs(bl) \cap I_{k_{\max}}^i) > 1$$

Finally:

$$\text{card}(Inputs(bl) \cap I_j^j) > 1$$

ii) Otherwise, for each $i + 1 \leq k \leq j - 1$, we have $bl \notin Coverage(in_k)$. Note that this is also the case if $j = i + 1$. In that case, we have:

$$\left(Inputs(bl) \cap I_j^i\right) \cup \{in_i\} = Inputs(bl) \cap I_i$$

where $I_i = I \setminus \{in_1, \ldots, in_{i-1}\}$. So:

$$\text{card}(Inputs(bl) \cap I_j^i) + 1 = \text{card}(Inputs(bl) \cap I_i)$$

Thus:

$$\text{card}(Inputs(bl) \cap I_j^j) = \text{card}(Inputs(bl) \cap I_i)$$

Because $[in_1, \ldots, in_i, \ldots, in_j, \ldots, in_n] \in ValidOrders(I)$, we have:

$$redundancy(in_i, I_i) > 0$$

So, because $bl \in Coverage(in_i)$, we have:

$$\text{card}(Inputs(bl) \cap I_i) > 1$$

Finally:

$$\mathrm{card}(Inputs(bl) \cap I_j^j) > 1$$

This concludes the proof by case for $bl \in Coverage(in_i) \setminus Coverage(in_j)$. Therefore, we proved by case that, for each $bl \in Coverage(in_i)$, we have:

$$\mathrm{card}(Inputs(bl) \cap I_j^j) > 1$$

Thus, we have $redundancy(in_i, I_j^j) > 0$.

Moreover, according to Lemma 5, $in_1, \ldots, in_i, \ldots, in_j, \ldots, in_n$ are distinct, so $in_i \in I_j^j$. Thus, $in_i \in Redundant(I_j^j)$.

Finally, we have from the second step (or the first one, if $j = i + 1$):

$$[in_1, \ldots, in_{i-1}, in_j, in_{i+1}, \ldots, in_{j-1}] \in ValidOrders(I)$$

Therefore, according to the definition of valid orders of removal steps (§ 3.3):

$$[in_1, \ldots, in_{i-1}, in_j, in_{i+1}, \ldots, in_{j-1}, in_i] \in ValidOrders(I)$$

4) Finally, if $j = n$ then the proof is complete. Otherwise, we prove by induction on $j \leq k \leq n$ that:

$$[in_1, \ldots, in_j, \ldots, in_i, in_{j+1}, \ldots, in_k] \in ValidOrders(I)$$

For the initialization $k = j$, we have from the third step:

$$[in_1, \ldots, in_j, \ldots, in_i] \in ValidOrders(I)$$

We now assume by induction hypothesis on $j \leq k < n$ that:

$$[in_1, \ldots, in_j, \ldots, in_i, in_{j+1}, \ldots, in_k] \in ValidOrders(I)$$

Because $[in_1, \ldots, in_i, \ldots, in_j, \ldots, in_n] \in ValidOrders(I)$, we have:

$$in_{k+1} \in Redundant(I \setminus \{in_1, \ldots, in_i, \ldots, in_j, \ldots, in_k\})$$

Moreover, we have:

$$\{in_1, \ldots, in_i, \ldots, in_j, \ldots, in_k\} = \{in_1, \ldots, in_j, \ldots, in_i, \ldots, in_k\}$$

So:

$$in_{k+1} \in Redundant(I \setminus \{in_1, \ldots, in_j, \ldots, in_i, \ldots, in_k\})$$

Thus, according to the definition of valid orders of removal steps (§ 3.3):

$$[in_1, \ldots, in_j, \ldots, in_i, in_{j+1}, \ldots, in_k, in_{k+1}] \in ValidOrders(I)$$

which concludes the induction step.

Therefore, we proved by induction the lemma:

$$[in_1, \ldots, in_j, \ldots, in_i, in_{j+1}, \ldots, in_n] \in ValidOrders(I)$$

□

The issue with our definition of the gain (§ 3.3) is that, to compute $gain(I)$ of an input set I, one has to try all the possible order of removal steps to determine which ones are valid. If I contains n inputs and we consider orders of $0 \leq k \leq n$ removal steps, there are $n \times \cdots \times (n - k + 1) = \frac{!n}{!(n-k)}$ possibilities to investigate, which is usually large.

Thus, we present an optimization to reduce the cost of computing the gain. First, we prove in Proposition 2 that, while the order of inputs matters to determine if an order of removal steps is valid, it does not matter anymore once we know the order is valid. In other words, inputs in a valid order of removal steps may be rearranged in any different order, and the rearranged order of removal steps would be valid. Formally, if $[in_1, \ldots, in_n]$ is a valid order of removal steps, then $[in_{\sigma(1)}, \ldots, in_{\sigma(n)}]$ is also a valid order, when σ denotes a permutation of the n inputs, i.e., the same inputs but (potentially) in a different order:

Proposition 2 (Permutation of a Valid Order)**.** *Let I be an input set. If $[in_1, \ldots, in_n] \in \mathit{ValidOrders}(I)$ and σ is a permutation on n elements, then $[in_{\sigma(1)}, \ldots, in_{\sigma(n)}] \in \mathit{ValidOrders}(I)$.*

Proof. Every permutation of a finite set can be expressed as the product of transpositions [25] (p.60). Let m be the number of such transpositions for σ. We prove the result by induction on m.

If $m = 0$, then σ is the identity and we have by hypothesis:

$$[in_{\sigma(1)}, \ldots, in_{\sigma(n)}] \in \mathit{ValidOrders}(I)$$

If $\sigma = \tau_{m+1} \circ \tau_m \circ \cdots \circ \tau_1$, then we denote $\sigma' = \tau_m \circ \cdots \circ \tau_1$. By induction hypothesis, we have:

$$[in_{\sigma'(1)}, \ldots, in_{\sigma'(n)}] \in \mathit{ValidOrders}(I)$$

Then, by applying Lemma 8 to the transposition $(i\,j) = \tau_{m+1}$, we have:

$$[in_{\tau_{m+1} \circ \sigma'(1)}, \ldots, in_{\tau_{m+1} \circ \sigma'(n)}] \in \mathit{ValidOrders}(I)$$

Finally, because $\sigma = \tau_{m+1} \circ \sigma'$, we conclude the induction step. □

Since the order of valid removal steps does not matter, we introduce a canonical order on removal steps. We assume each input is numbered, and we focus on steps where inputs are always removed in increasing order. Formally, $[in_{i_1}, \ldots, in_{i_m}]$ is in *canonical order* if $1 \leq i_1 < \cdots < i_m \leq n$. For instance, $[in_2, in_4, in_7]$ is in canonical order, but not $[in_4, in_2, in_7]$. That way, to investigate which orders of removal steps are valid, we can focus on inputs in increasing order of index, instead of considering backtracking on previous inputs.

Theorem 1 (Canonical Order)**.** *Let $I = \{in_1, \ldots, in_n\}$ be an input set. There exists $[in_{i_1}, \ldots, in_{i_m}] \in \mathit{ValidOrders}(I)$ such that $1 \leq i_1 < \cdots < i_m \leq n$ and:*

$$\sum_{1 \leq j \leq m} \mathit{cost}(in_{i_j}) = \mathit{gain}(I)$$

Proof. Let $[in_{i'_1}, \ldots, in_{i'_m}] \in \mathit{ValidOrders}(I)$ be a valid order of removal steps with a maximal cumulative cost i.e., according to § 3.3:

$$\sum_{1 \leq j \leq m} \mathit{cost}(in_{i'_j}) = \mathit{gain}(I)$$

If $m = 0$, then $[\,]$ satisfies the theorem for a gain $= 0$.

Otherwise, we assume $m > 0$. According to Lemma 5, $[in_{i'_1}, \ldots, in_{i'_m}]$ contains m distinct inputs. Let $\sigma \in S_m$ be the permutation such that:

$$[in_{\sigma(i'_1)}, \ldots, in_{\sigma(i'_m)}] = [in_{i_1}, \ldots, in_{i_m}]$$

with $1 \leq i_1 < \cdots < i_m \leq n$.

According to Proposition 2, we have $[in_{i_1}, \ldots, in_{i_m}] \in \mathit{ValidOrders}(I)$.

Moreover, because a permutation of the elements of a sum does not change the value of the sum, we have:

$$\sum_{1 \leq j \leq m} \mathit{cost}(in_{i_j}) = \sum_{1 \leq j \leq m} \mathit{cost}(in_{i'_j}) = \mathit{gain}(I)$$

Hence the theorem. □

Theorem 1 allows us to focus on inputs in increasing order, instead of exploring all possible input orders. Thus, this optimization saves computation steps and makes computing the gain more tractable. More precisely, since there are $!k$ ways of ordering k selected inputs, there are "only" $\frac{!n}{!k!(n-k)}$ remaining possibilities to investigate, instead of $\frac{!n}{!(n-k)}$.

B FORMAL RESULTS FOR THE SEARCH

We prove in Appendix B.1 Proposition 3 and Theorem 2 presented in § 7.5 on local dominance, in Appendix B.2 Proposition 8 and Theorem 3 presented in § 7.6 on dividing the problem, and in Appendix B.3 desirable properties of roofers (Theorem 4) and misers (Theorem 5) during the genetic search (Section 8).

B.1 Local Dominance

In this section we prove, as presented in § 7.5, that the local dominance relation is local (Proposition 3), hence is faithful to its name, and that non locally-dominated inputs, as a whole, locally-dominate all the locally dominated inputs (Theorem 2). We start by the locality property.

Proposition 3 (Local Dominance is Local)**.** *Let $in_1 \in I_{search}$ be a remaining input and $S \subseteq I_{search}$ be a subset of the remaining inputs. If $in_1 \sqsubseteq S$ then $in_1 \sqsubseteq S \cap \{in_2 \in I_{search} \mid in_1 \sqcap in_2\}$.*

Proof. We assume $in_1 \sqsubseteq S$. So, by definition of local dominance (§ 7.5), we have $in_1 \notin S$, $Coverage(in_1) \subseteq Coverage(S)$, and $cost(in_1) \geq cost(S)$. First, because $in_1 \notin S$, we have $in_1 \notin S \cap \{in_2 \in I_{search} \mid in_1 \sqcap in_2\}$.

Moreover, for every $in_2 \in I_{search}$, if there is no overlap between in_1 and in_2 then, according to the overlapping relation (§ 7.5), we have $Coverage(in_1) \cap Coverage(in_2) = \varnothing$. Thus:

$$Coverage(in_1) \cap Coverage(S \cap \{in_2 \in I_{search} \mid \neg in_1 \sqcap in_2\}) = \varnothing$$

Hence, because $Coverage(in_1) \subseteq Coverage(S)$, we have:

$$\begin{aligned} Coverage(in_1) &= Coverage(in_1) \cap Coverage(S) \\ &= Coverage(in_1) \cap (Coverage(S \cap \{in_2 \in I_{search} \mid in_1 \sqcap in_2\}) \\ &\qquad\qquad \cup\ Coverage(S \cap \{in_2 \in I_{search} \mid \neg in_1 \sqcap in_2\})) \\ &= (Coverage(in_1) \cap Coverage(S \cap \{in_2 \in I_{search} \mid in_1 \sqcap in_2\})) \\ &\quad \cup (Coverage(in_1) \cap Coverage(S \cap \{in_2 \in I_{search} \mid \neg in_1 \sqcap in_2\})) \\ &= (Coverage(in_1) \cap Coverage(S \cap \{in_2 \in I_{search} \mid in_1 \sqcap in_2\})) \end{aligned}$$

So, $Coverage(in_1) \subseteq Coverage(S \cap \{in_2 \in I_{search} \mid in_1 \sqcap in_2\})$.

Finally, $cost(S) \geq cost(S \cap \{in_2 \in I_{search} \mid in_1 \sqcap in_2\})$. Thus, because $cost(in_1) \geq cost(S)$, we have $cost(in_1) \geq cost(S \cap \{in_2 \in I_{search} \mid in_1 \sqcap in_2\})$.

Therefore, $in_1 \sqsubseteq S \cap \{in_2 \in I_{search} \mid in_1 \sqcap in_2\}$. □

We now prove that the local dominance relation is asymmetric (Proposition 5). First, because an input always covers some objectives, it can be locally dominated only by a non-empty input subset.

Lemma 9 (Non-Empty Local Dominance)**.** *Let $in \in I_{search}$ be an input and $S \subseteq I_{search}$ be a subset of the remaining inputs. If $in \sqsubseteq S$ then $S \neq \varnothing$.*

Proof. The proof is done by contradiction. If $S = \varnothing$ then $Coverage(S) = \varnothing$. Because $in \sqsubseteq S$ we have $Coverage(in) \subseteq Coverage(S)$. So, $Coverage(in) = \varnothing$, which contradicts Lemma 1. □

Second, when two inputs locally dominates each other, they are equivalent in the sense of the equivalence relation from § 7.4, that we denote $\equiv$ in the following. But, because we removed duplicates in § 7.4, this case can happen only if the inputs are the same, which is excluded by § 7.5.

Lemma 10 (Local Dominance for Singletons is Asymmetric). *Let $in_1, in_2 \in I_{search}$ be two inputs. If $in_1 \sqsubseteq \{in_2\}$, then $in_2 \not\sqsubseteq \{in_1\}$.*

Proof. We assume by contradiction that $in_1 \sqsubseteq \{in_2\}$ and $in_2 \sqsubseteq \{in_1\}$.

Hence, we have: 1) $Coverage(in_1) \subseteq Coverage(in_2)$ and $Coverage(in_2) \subseteq Coverage(in_1)$, so $Coverage(in_1) = Coverage(in_2)$. 2) $cost(in_1) \geq cost(in_2)$ and $cost(in_2) \geq cost(in_1)$, so $cost(in_1) = cost(in_2)$. So, we have $in_1 \equiv in_2$.

Because we removed duplicates in § 7.4, we have $in_1 = in_2$. So, $in_1 \sqsubseteq \{in_1\}$, which contradicts in § 7.5 that an input cannot locally dominates itself. □

Note that this result is also useful to prove transitivity (Proposition 7).

Third, because the cost of an input is positive, if an input is locally dominated by several inputs, then they have a strictly smaller cost.

Lemma 11 (Cost Hierarchy). *Let $in_1 \in I_{search}$ be an input and $S \subseteq I_{search}$ be a subset of the remaining inputs. If $in_1 \sqsubseteq S$ and $\mathrm{card}(S) \geq 2$ then for every $in_2 \in S$ we have $cost(in_2) < cost(in_1)$.*

Proof. Because $in_1 \sqsubseteq S$ we have $cost(in_1) \geq cost(S)$. So, for every $in_2 \in S$ we have $cost(in_2) \leq cost(in_1)$. If $\mathrm{card}(S) \geq 2$, then the inequality is strict because, according to Lemma 2, the cost is positive $cost(in_2) > 0$. □

Finally, we prove as expected that the local dominance relation is asymmetric.

Proposition 4 (Local Dominance for Subsets is Asymmetric). *Let $in_1, in_2 \in I_{search}$ be two inputs and $S_1, S_2 \subseteq I_{search}$ be two subsets of the remaining inputs. If $in_1 \sqsubseteq S_1$, $in_2 \in S_1$, and $in_2 \sqsubseteq S_2$, then $in_1 \notin S_2$.*

Proof. The proof is made by contradiction. We assume $in_1 \in S_2$ and prove a contradiction in different cases for $\mathrm{card}(S_1)$ and $\mathrm{card}(S_2)$.

$\mathrm{card}(S_1) = 0$ or $\mathrm{card}(S_2) = 0$ are not possible, because this would contradict Lemma 9.

If $\mathrm{card}(S_1) = 1$ and $\mathrm{card}(S_2) = 1$, then $S_1 = \{in_2\}$ and $S_2 = \{in_1\}$. Thus, $in_1 \sqsubseteq \{in_2\}$ and $in_2 \sqsubseteq \{in_1\}$, which contradicts Lemma 10.

If $\mathrm{card}(S_1) = 1$ then $S_1 = \{in_2\}$. Because $in_1 \sqsubseteq S_1$, we have $cost(in_1) \geq cost(in_2)$. Moreover, because $in_2 \sqsubseteq S_2$ and $in_1 \in S_2$, if $\mathrm{card}(S_2) \geq 2$ then according to Lemma 11 we have $cost(in_1) < cost(in_2)$, hence the contradiction.

Because $in_1 \sqsubseteq S_1$ and $in_2 \in S_1$, if $\mathrm{card}(S_1) \geq 2$ then according to Lemma 11 we have $cost(in_2) < cost(in_1)$. Because $in_2 \sqsubseteq S_2$ and $in_1 \in S_2$, if $\mathrm{card}(S_2) \geq 2$ then according to Lemma 11 we have $cost(in_1) < cost(in_2)$. Hence the contradiction $cost(in_1) < cost(in_1)$. □

Based on our definition of local dominance for subsets (§ 7.5), we introduce the corresponding definition for inputs, in order to state more easily Corollary 1, then prove Theorem 2.

Definition 2 (Local Dominance). The input $in_1 \in I_{search}$ *locally-dominates* the input $in_2 \in I_{search}$, denoted $in_1 \hookrightarrow in_2$, if there exists a subset $S \subseteq I_{search}$ such that $in_1 \in S$ and $in_2 \sqsubseteq S$.

Proposition 5 (Local Dominance for Inputs is Asymmetric). *The $\hookrightarrow$ relation is asymmetric i.e., for every $in_1, in_2 \in I_{search}$, if $in_2 \hookrightarrow in_1$, then $in_1 \not\hookrightarrow in_2$.*

Proof. If $in_2 \hookrightarrow in_1$ then there exists $S_1 \subseteq I_{search}$ such that $in_2 \in S_1$ and $in_1 \sqsubseteq S_1$. The proof is done by contradiction, assuming $in_1 \hookrightarrow in_2$. Hence, there exists $S_2 \subseteq I_{search}$ such that $in_1 \in S_2$ and $in_2 \sqsubseteq S_2$ According to Proposition 4 we have $in_1 \notin S_2$, hence the contradiction. □

We now prove that the local dominance relation is transitive (Proposition 7).

Proposition 6 (Local Dominance for Subsets is Transitive). *Let $in_1, in_2 \in I_{search}$ be two inputs and $S_1, S_2 \subseteq I_{search}$ be two subsets of the remaining inputs.*

If $in_1 \sqsubseteq S_1$, $in_2 \in S_1$, and $in_2 \sqsubseteq S_2$, then $in_1 \sqsubseteq (S_1 \setminus \{in_2\}) \cup S_2$.

Proof. Because $Coverage(in_1) \subseteq Coverage(S_1)$, $in_2 \in S_1$, and $Coverage(in_2) \subseteq Coverage(S_2)$, we have:

$$\begin{aligned} Coverage(in_1) &\subseteq Coverage(S_1 \setminus \{in_2\}) \cup Coverage(S_2) \\ &= Coverage((S_1 \setminus \{in_2\}) \cup S_2) \end{aligned}$$

Moreover, because $cost(in_1) \geq cost(S_1)$, $in_2 \in S_1$, and $cost(in_2) \geq cost(S_2)$, we have:

$$\begin{aligned} cost(in_1) &\geq cost(S_1 \setminus \{in_2\}) + cost(S_2) \\ &\geq cost((S_1 \setminus \{in_2\}) \cup S_2) \end{aligned}$$

Finally, because $in_1 \sqsubseteq S_1$ we have $in_1 \notin S_1$. We prove $in_1 \notin (S_1 \setminus \{in_2\}) \cup S_2$ by contradiction. If $in_1 \in (S_1 \setminus \{in_2\}) \cup S_2$ then, because $in_1 \notin S_1$, we have $in_1 \in S_2$. In that case, we have

$$\begin{aligned} cost(in_1) &\geq cost(S_1 \setminus \{in_2\}) + cost(S_2) \\ &= cost(S_1 \setminus \{in_2\}) + cost(S_2 \setminus \{in_1\}) + cost(in_1) \\ &\geq cost(in_1) \end{aligned}$$

Hence, by subtracting $cost(in_1)$, we have:

$$cost(S_1 \setminus \{in_2\}) + cost(S_2 \setminus \{in_1\}) = 0$$

Thus, according to Lemma 2, we have $S_1 = \{in_2\}$ and $S_2 = \{in_1\}$. Therefore $in_1 \sqsubseteq \{in_2\}$ and $in_2 \sqsubseteq \{in_1\}$, which contradicts Lemma 10. □

Proposition 7 (Local Dominance for Inputs is Transitive). *The $\hookrightarrow$ relation is transitive i.e., for every $in_1, in_2, in_3 \in I_{search}$, if $in_3 \hookrightarrow in_2$ and $in_2 \hookrightarrow in_1$, then $in_3 \hookrightarrow in_1$.*

Proof. If $in_3 \hookrightarrow in_2$ and $in_2 \hookrightarrow in_1$, then there exists $S_1, S_2 \subseteq I_{search}$ such that $in_2 \in S_1$, $in_1 \sqsubseteq S_1$, $in_3 \in S_2$, and $in_2 \sqsubseteq S_2$. According to Proposition 6, we have $in_1 \sqsubseteq (S_1 \setminus \{in_2\}) \cup S_2$. Hence, because $in_3 \in S_2 \subseteq (S_1 \setminus \{in_2\}) \cup S_2$ we have $in_3 \hookrightarrow in_1$. □

We finally prove that the local dominance relation is acyclic.

Corollary 1 (Local Dominance is Acyclic). *There exists no cycle $in_1 \hookrightarrow \ldots \hookrightarrow in_n \hookrightarrow in_1$ such that $in_1, \ldots, in_n \in I_{search}$.*

Proof. The proof is done by contradiction, by assuming the existence of such a cycle $in_1 \hookrightarrow \ldots \hookrightarrow in_n \hookrightarrow in_1$. According to Proposition 7, we have by transitivity that $in_1 \hookrightarrow in_1$. Then, according to Proposition 5, we have by asymmetry $in_1 \not\hookrightarrow in_1$. Hence the contradiction. □

Finally, we use the transitivity (Proposition 7 and Proposition 6) and the acyclicity (Corollary 1) of the local-dominance relation to prove that non locally-dominated inputs, as a whole, locally-dominate all the locally dominated inputs.

Theorem 2 (Local Dominance Hierarchy). *For every locally-dominated input $in \in I_{search}$, there exists a subset $S \subseteq I_{search}$ of not locally-dominated inputs such that $in \sqsubseteq S$.*

Proof. For each input $in_0 \in I_{search}$ we denote:

$$LocDoms(in_0) \stackrel{\text{def}}{=} \{in \in I_{search} \mid in \hookrightarrow in_0\}$$

the input set which locally dominate in_0.

Let $in_0 \in I_{search}$ be an input. We prove by induction on $LocDoms(in_0)$ that either in_0 is not locally dominated or there exists a subset $S \subseteq I_{search}$ of not locally-dominated inputs such that $in_0 \sqsubseteq S$.

For the initialization, we consider $LocDoms(in_0) = \varnothing$. In that case, in_0 is not locally dominated.

For the induction step, we consider $LocDoms(in_0) \neq \varnothing$, so in_0 is a locally dominated input. Hence, there exists $S_0 \subseteq I_{search}$ such that $in_0 \sqsubseteq S_0$. We consider two cases.

Either every input in S_0 is not locally dominated. In that case $S = S_0$ satisfies the inductive property.

Or there exists $n \geq 1$ input $in_1, \ldots, in_n$ in S_0 which are locally dominated. The rest of the proof is done in two steps.

First, let $1 \leq i \leq n$. We prove that $LocDoms(in_i) \subset LocDoms(in_0)$, with a strict inclusion.

Because $in_i \in S_0$ and $in_0 \sqsubseteq S_0$, we have $in_i \hookrightarrow in_0$. Hence, $in_i \in LocDoms(in_0)$.

Let $in \in LocDoms(in_i)$, so we have $in \hookrightarrow in_i$. So, according to Proposition 7, we have by transitivity $in \hookrightarrow in_0$, thus $in \in LocDoms(in_0)$. Hence, $LocDoms(in_i) \subseteq LocDoms(in_0)$.

According to Corollary 1, there is no cycle so $in_i \not\hookrightarrow in_i$. Hence, $in_i \notin LocDoms(in_i)$.

Finally, we have $LocDoms(in_i) \subseteq LocDoms(in_0)$ and $in_i \in LocDoms(in_0) \setminus LocDoms(in_i)$. Therefore $LocDoms(in_i) \subset LocDoms(in_0)$.

Because $LocDoms(in_i) \subset LocDoms(in_0)$, we can apply the induction hypothesis on in_i, which is locally dominated. Therefore, for each $1 \leq i \leq n$, there exists a subset $S_i \subseteq I_{search}$ of not locally-dominated inputs such that $in_i \sqsubseteq S_i$.

Second, we use these S_i to define by induction on $0 \leq m < n$ the following input sets:

$$
\begin{aligned}
I_0 &= S_0 \\
I_{m+1} &= (I_m \setminus \{in_{m+1}\}) \cup S_{m+1}
\end{aligned}
$$

and we prove by induction on $0 \leq m \leq n$ that $in_0 \sqsubseteq I_m$ and that either, $m < n$ and $in_{m+1}, \ldots, in_n$ are the only locally dominated inputs in I_m, or $m = n$ and I_m contains no locally dominated input.

For the initialization, we have $I_0 = S_0$ and the inductive property is satisfied because $in_0 \sqsubseteq S_0$ and $in_1, \ldots, in_n$ are the only locally dominated inputs in S_0.

For the induction step $m + 1 \leq n$, we assume that $in_0 \sqsubseteq I_m$ and that $in_{m+1}, \ldots, in_n$ are the only locally dominated inputs in I_m.

Because $I_{m+1} = (I_m \setminus \{in_{m+1}\}) \cup S_{m+1}$ and S_{m+1} contains no locally-dominated input, we have that either, $m + 1 < n$ and $in_{m+2}, \ldots, in_n$ are the only locally dominated inputs in I_{m+1}, or $m + 1 = n$ and I_{m+1} contains no locally-dominated input.

Moreover, because $in_0 \sqsubseteq I_m$, $in_{m+1} \in I_m$, and $in_{m+1} \sqsubseteq S_{m+1}$, then according to Proposition 6, we have $in_0 \sqsubseteq (I_m \setminus \{in_{m+1}\}) \cup S_{m+1}$. Hence $in_0 \sqsubseteq I_{m+1}$, which concludes the induction step.

Therefore, $in_0 \sqsubseteq I_n$ and I_n contains no locally-dominated input. Thus, $S = I_n$ satisfies the inductive property on $LocDoms(in_0)$. Hence the claim for not locally-dominated inputs. □

B.2 Dividing the Problem

In this section, we prove that redundancy updates are local (Lemma 12), that reductions on different overlapping components can be performed independently (Proposition 8), and finally that the gain can be independently computed on each overlapping component (Theorem 3). The main purpose of the section is to justify we can divide our problem into subproblems (§ 7.6). We start by the locality.

Lemma 12 (Redundancy Updates are Local)**.** *Let I be an input set and let $in_1, in_2 \in I$.*
If $redundancy(in_2, I \setminus \{in_1\}) \neq redundancy(in_2, I)$, then $in_1 \sqcap in_2$.

Proof. The proof is done by contraposition. If in_1 and in_2 do not overlap, then $Coverage(in_1) \cap Coverage(in_2) = \varnothing$. So, for every $bl \in Coverage(in_2)$, we have $in_1 \notin Inputs(bl)$, and thus $Inputs(bl) \cap (I \setminus \{in_1\}) = Inputs(bl) \cap I$. Therefore, $redundancy(in_2, I \setminus \{in_1\}) = redundancy(in_2, I)$. □

Then, we prove the independence of removal steps performed on different components.

Lemma 13 (Independent Redundancies). *Let I be an input set. For each overlapping component $C \in OverlapComps(I)$, for each input $in_0 \in C$, and for each order of removal steps $[in_1, \dots, in_n]$, if $in_1, \dots, in_n \notin C$, then $redundancy(in_0, I \setminus \{in_1, \dots, in_n\}) = redundancy(in_0, I)$.*

Proof. $OverlapComps(I)$ are the connected components for the redundant inputs in I, hence they are disjoint. Therefore, for each $in_0 \in C$, because $in_1, \dots, in_n \notin C$, we have that in_0 do not overlap with any of the $in_1, \dots, in_n$. Therefore, according to Lemma 12, we have $redundancy(in_0, I) = redundancy(in_0, I \setminus \{in_1\}) = \cdots = redundancy(in_0, I \setminus \{in_1, \dots, in_n\})$. □

Proposition 8 (Independent Removal Steps).

Let I be an input set and let $C_1, \dots, C_c \in OverlapComps(I)$ denote c overlapping components. If, for each $0 \le i \le c$, there exists inputs $in^i_1, \dots, in^i_{n_i} \in C_i$ such that $[in^i_1, \dots, in^i_{n_i}] \in ValidOrders(I)$, then:

$$[in^1_1, \dots, in^1_{n_1}] + \cdots + [in^c_1, \dots, in^c_{n_c}] \in ValidOrders(I)$$

where + denotes list concatenation.

Proof. The proof is done by induction on c.

If $c = 0$, then $[in^1_1, \dots, in^1_{n_1}] + \cdots + [in^c_1, \dots, in^c_{n_c}] = [\,] \in ValidOrders(I)$.

We assume by induction that $[in^1_1, \dots, in^1_{n_1}] + \cdots + [in^c_1, \dots, in^c_{n_c}] \in ValidOrders(I)$.

We denote ℓ this order of removal steps and for the sake of simplicity we also denote $S = \{in^1_1, \dots, in^1_{n_1}, \dots, in^c_1, \dots, in^c_{n_c}\}$.

By induction, let $C_{c+1} \in OverlapComps(I)$ be another overlapping component and let $in^{c+1}_1, \dots, in^{c+1}_{n_{c+1}} \in C_{c+1}$ such that $[in^{c+1}_1, \dots, in^{c+1}_{n_{c+1}}] \in ValidOrders(I)$.

We now prove that $\ell + [in^{c+1}_1, \dots, in^{c+1}_{n_{c+1}}] \in ValidOrders(I)$.

This is done by proving by induction on $0 \le j \le n_{c+1}$ that:

$$\ell + [in^{c+1}_1, \dots, in^{c+1}_j] \in ValidOrders(I)$$

and that, for each $in_0 \in C_{c+1} \setminus S_j$, we have:

$$redundancy(in_0, I \setminus (S \cup S_j)) = redundancy(in_0, I \setminus S_j)$$

where $S_j = \{in^{c+1}_1, \dots, in^{c+1}_j\}$.

If $n_{c+1} = 0$, then $\ell + [in^{c+1}_1, \dots, in^{c+1}_{n_{c+1}}] = \ell \in ValidOrders(I)$. Moreover, because C_{c+1} is disjoint with $C_1, \dots, C_c$, according to Lemma 13, performing the ℓ removal steps does not change the redundancies of inputs in C_{c+1} i.e., for each $in_0 \in C_{c+1}$, we have $redundancy(in_0, I \setminus S) = redundancy(in_0, I)$.

We now assume by induction that $\ell + [in^{c+1}_1, \dots, in^{c+1}_j] \in ValidOrders(I)$ and for each $in_0 \in C_{c+1} \setminus S_j$, we have $redundancy(in_0, I \setminus (S \cup S_j)) = redundancy(in_0, I \setminus S_j)$.

Because $[in^{c+1}_1, \dots, in^{c+1}_{n_{c+1}}] \in ValidOrders(I)$, we have $in^{c+1}_{j+1} \in Redundant(I \setminus S_j)$. Moreover, $in^{c+1}_{j+1} \in C_{c+1}$ hence $in^{c+1}_{j+1} \notin S$. Therefore, using the induction hypothesis with $in_0 = in^{c+1}_{j+1}$, we have $in^{c+1}_{j+1} \in Redundant(I \setminus (S \cup S_j))$. Therefore, according to our definition of valid removal steps (§ 3.3), $\ell + [in^{c+1}_1, \dots, in^{c+1}_{j+1}] \in ValidOrders(I)$.

We denote $S_{j+1} = S_j \cup \{in^{c+1}_{j+1}\}$. Let $in_0 \in C_{c+1} \setminus S_{j+1}$.

To complete the induction step, we prove that:

$$redundancy(in_0, I \setminus (S \cup S_{j+1})) = redundancy(in_0, I \setminus S_{j+1})$$

We remind that, for each input set X, we have:

$$redundancy(in_0, X) = \min\{\mathrm{card}(Inputs(bl) \cap X) \mid bl \in Coverage(in_0)\} - 1$$

We denote as critical the objectives contributing to the redundancy:

$$CritSubCls(in_0, X) \overset{\text{def}}{=} \arg\min\{\mathrm{card}(Inputs(bl) \cap X) \mid bl \in Coverage(in_0)\}$$

Because $in_0 \in C_{c+1}$, we know that in_0 does not overlap with inputs in S so, for each $bl \in Coverage(in_0)$, we have $Inputs(bl) \cap (I \setminus S) = Inputs(bl) \cap I$. Thus, by removing inputs of S_j from both sides, we have $Inputs(bl) \cap (I \setminus (S \cup S_j)) = Inputs(bl) \cap (I \setminus S_j)$. Therefore:

$$CritSubCls(in_0, I \setminus (S \cup S_j)) = CritSubCls(in_0, I \setminus S_j)$$

We consider two cases:

1) Either there exists $bl \in CritSubCls(in_0, I \setminus S_j)$ such that $in_{j+1}^{c+1} \in Inputs(bl)$. In that case:

$$\mathrm{card}(Inputs(bl) \cap (I \setminus (S \cup S_{j+1}))) = \mathrm{card}(Inputs(bl) \cap (I \setminus (S \cup S_j))) - 1$$

$$\mathrm{card}(Inputs(bl) \cap (I \setminus S_{j+1})) = \mathrm{card}(Inputs(bl) \cap (I \setminus S_j)) - 1$$

Because bl is critical and a redundancy can decrease at most by 1 after a reduction (Lemma 4) so the cardinalities for other objectives cannot decrease below the previous redundancy minus one, we have:

$$redundancy(in_0, I \setminus (S \cup S_{j+1})) = redundancy(in_0, I \setminus (S \cup S_j)) - 1$$

$$redundancy(in_0, I \setminus S_{j+1}) = redundancy(in_0, I \setminus S_j) - 1$$

2) Or for each $bl \in CritSubCls(in_0, I \setminus S_j)$, we have $in_{j+1}^{c+1} \notin Inputs(bl)$. In that case:

$$\mathrm{card}(Inputs(bl) \cap (I \setminus (S \cup S_{j+1}))) = \mathrm{card}(Inputs(bl) \cap (I \setminus (S \cup S_j)))$$

$$\mathrm{card}(Inputs(bl) \cap (I \setminus S_{j+1})) = \mathrm{card}(Inputs(bl) \cap (I \setminus S_j))$$

Because bl is critical and a redundancy can decrease at most by 1 after a reduction (Lemma 4) so the cardinalities for non-critical objectives cannot decrease below the previous redundancy we have:

$$redundancy(in_0, I \setminus (S \cup S_{j+1})) = redundancy(in_0, I \setminus (S \cup S_j))$$

$$redundancy(in_0, I \setminus S_{j+1}) = redundancy(in_0, I \setminus S_j)$$

In both cases, by induction hypothesis $redundancy(in_0, I \setminus (S \cup S_j)) = redundancy(in_0, I \setminus S_j)$, hence we have $redundancy(in_0, I \setminus (S \cup S_{j+1})) = redundancy(in_0, I \setminus S_{j+1})$.

This completes the induction on $0 \leq j \leq n_{c+1}$. In particular, we proved that $\ell + [in_1^{c+1}, \ldots, in_{n_{c+1}}^{c+1}] \in ValidOrders(I)$, which completes the induction on c, hence the result. □

We now prove that the gain can be independently computed on each overlapping component (Theorem 3). To do so, we have to prove intermediate results, including Proposition 9.

Lemma 14 (Redundancy within an Overlapping Component). *Let I be an input set. For each overlapping component $C \in OverlapComps(I)$, for each input $in_0 \in C$, and for each order of removal steps $[in_1, \ldots, in_n]$, if $in_1, \ldots, in_n \in C$, then:*

$$redundancy(in_0, C \setminus \{in_1, \ldots, in_n\}) = redundancy(in_0, I \setminus \{in_1, \ldots, in_n\})$$

Proof. We remind that, for each input set X, we have:

$$redundancy(in_0, X) = \min\{\mathrm{card}(Inputs(bl) \cap X) \mid bl \in Coverage(in_0)\} - 1$$

For the sake of simplicity, we denote $X_C = C \setminus \{in_1, \ldots, in_n\}$ and $X_I = I \setminus \{in_1, \ldots, in_n\}$. Let $in_0 \in C$ and $bl \in Coverage(in_0)$.

Because $C \in OverlapComps(I)$, we have $C \subseteq I$, hence $X_C \subseteq X_I$. So $Inputs(bl) \cap X_C \subseteq Inputs(bl) \cap X_I$. Moreover, for each $in \in Inputs(bl) \cap X_I$ we have $bl \in Coverage(in_0) \cap Coverage(in)$, so $in_0 \sqcap in$, and thus $in \in X_C$.

Therefore, $Inputs(bl) \cap X_C = Inputs(bl) \cap X_I$.

Hence the result $redundancy(in_0, X_C) = redundancy(in_0, X_I)$. □

Corollary 2 (Component Validity). *Let I be an input set. For each overlapping component $C \in OverlapComps(I)$ and for each order of removal steps $[in_1, \dots, in_n]$, if $[in_1, \dots, in_n] \in ValidOrders(C)$, then $[in_1, \dots, in_n] \in ValidOrders(I)$.*

Proof. The proof is done by induction on n.

For the initialization $n = 0$, we have $[in_1, \dots, in_n] = [\,] \in ValidOrders(I)$.

For the induction step, we consider $[in_1, \dots, in_n, in_{n+1}] \in ValidOrders(C)$ and we assume by induction that $[in_1, \dots, in_n] \in ValidOrders(I)$.

Because $[in_1, \dots, in_n, in_{n+1}] \in ValidOrders(C)$, according to our definition of redundant inputs and removal steps (§ 3.3), we have $in_1, \dots, in_n \in C$. Hence, according to Lemma 14, we have for each $in_0 \in C$:

$$redundancy(in_0, C \setminus \{in_1, \dots, in_n\}) = redundancy(in_0, I \setminus \{in_1, \dots, in_n\})$$

Thus, because $C \subseteq I$, according to § 3.3 we have:

$$Redundant(C \setminus \{in_1, \dots, in_n\}) \subseteq Redundant(I \setminus \{in_1, \dots, in_n\})$$

Because $[in_1, \dots, in_n, in_{n+1}] \in ValidOrders(C)$, we have $in_{n+1} \in Redundant(C \setminus \{in_1, \dots, in_n\})$ by definition of removal steps (§ 3.3). Thus, $in_{n+1} \in Redundant(I \setminus \{in_1, \dots, in_n\})$.

By induction hypothesis $[in_1, \dots, in_n] \in ValidOrders(I)$.

Therefore $[in_1, \dots, in_n, in_{n+1}] \in ValidOrders(I)$, which completes the induction step. □

Proposition 9 (Reductions withing an Overlapping Component). *Let I be an input set. For each overlapping component $C \in OverlapComps(I)$ and for each order of removal steps $[in_1, \dots, in_n]$, if $[in_1, \dots, in_n] \in ValidOrders(I)$ and $in_1, \dots, in_n \in C$, then $[in_1, \dots, in_n] \in ValidOrders(C)$.*

Proof. The proof is done by induction on n.

If $n = 0$ then $[in_1, \dots, in_n] = [\,] \in ValidOrders(C)$.

Otherwise, we consider $[in_1, \dots, in_n, in_{n+1}]$ with $in_1, \dots, in_n, in_{n+1} \in C$ and we assume by induction that $[in_1, \dots, in_n] \in ValidOrders(C)$.

We assume $[in_1, \dots, in_n, in_{n+1}] \in ValidOrders(I)$, so $in_{n+1} \in Redundant(I \setminus \{in_1, \dots, in_n\})$.

According to Lemma 14 applied to inputs $in_0 \in C \setminus \{in_1, \dots, in_n\}$ with redundancy > 0, we have:

$$Redundant(C \setminus \{in_1, \dots, in_n\}) = Redundant(I \setminus \{in_1, \dots, in_n\})$$

Hence, $in_{n+1} \in Redundant(C \setminus \{in_1, \dots, in_n\})$.

Thus, because $[in_1, \dots, in_n] \in ValidOrders(C)$, according to our definition of valid removal steps (§ 3.3), we have the result $[in_1, \dots, in_n, in_{n+1}] \in ValidOrders(C)$. □

Finally, we conclude the section by proving the claim on Theorem 3, which is used to divide our problem into subproblems (§ 7.6).

Theorem 3 (Divide the Gain). *For each input set I:*

$$gain(I) = \sum_{C \in OverlapComps(I)} gain(C)$$

Proof. Let $[in_1, \dots, in_n] \in ValidOrders(I)$ be a valid order of removal steps such that its cumulative cost $\sum_{1 \leq i \leq n} cost(in_i) = gain(I)$ is maximal (§ 3.3), and let $OverlapComps(I) = \{C_1, \dots, C_c\}$ denote the overlapping components, without a particular order.

We denote $[in_1, \dots, in_n]_{C_j}$ the largest sublist (Definition 1) of $[in_1, \dots, in_n]$ containing only inputs in the overlapping component C_j. Because $[in_1, \dots, in_n] \in ValidOrders(I)$, according to Lemma 7 we have $[in_1, \dots, in_n]_{C_j} \in ValidOrders(I)$ as well. So, according to Proposition 8:

$$[in_1, \dots, in_n]_{C_1} + \dots + [in_1, \dots, in_n]_{C_c} \in ValidOrders(I)$$

Because the overlapping components form a partition of the redundant inputs, the cumulative cost of this order of removal steps is the same as $[in_1, \ldots, in_n]$:

$$\sum_{1 \leq j \leq c} \sum_{in \in [in_1, \ldots, in_n]_{C_j}} cost(in) = \sum_{1 \leq i \leq n} cost(in_i) = gain(I)$$

We now consider any overlapping component C_{j_1} and we prove that:

$$\sum_{in \in [in_1, \ldots, in_n]_{C_{j_1}}} cost(in) = gain(C_{j_1})$$

The proof is done in two steps.

First, note that because $[in_1, \ldots, in_n]_{C_{j_1}} \in ValidOrders(I)$ and contains only inputs in C_{j_1}, according to Proposition 9 we have $[in_1, \ldots, in_n]_{C_{j_1}} \in ValidOrders(C_{j_1})$ as well.

Second, we prove by contradiction that $[in_1, \ldots, in_n]_{C_{j_1}}$ has a maximal cumulative cost in C_{j_1}. We assume by contradiction that there exists a valid order of removal steps $[in'_1, \ldots, in'_{n'}] \in ValidOrders(C_{j_1})$ such that:

$$\sum_{in \in [in'_1, \ldots, in'_{n'}]} cost(in) > \sum_{in \in [in_1, \ldots, in_n]_{C_{j_1}}} cost(in)$$

Because $[in'_1, \ldots, in'_{n'}] \in ValidOrders(C_{j_1})$, according to Corollary 2 we have $[in'_1, \ldots, in'_{n'}] \in ValidOrders(I)$ as well. So, according to Proposition 8, we have:

$$[in_1, \ldots, in_n]_{C_1} + \cdots + [in_1, \ldots, in_n]_{C_{j_1 - 1}} + [in'_1, \ldots, in'_{n'}]$$

$$+ [in_1, \ldots, in_n]_{C_{j_1 + 1}} + \cdots + [in_1, \ldots, in_n]_{C_c} \in ValidOrders(I)$$

The cumulative cost of this valid order of removal steps is thus larger than the cumulative cost of $[in_1, \ldots, in_n]_{C_1} + \cdots + [in_1, \ldots, in_n]_{C_c}$:

$$\sum_{in \in [in'_1, \ldots, in'_{n'}]} cost(in) + \sum_{1 \leq j \leq c \wedge j \neq j_1} \sum_{in \in [in_1, \ldots, in_n]_{C_j}} cost(in)$$

$$> \sum_{1 \leq j \leq c} \sum_{in \in [in_1, \ldots, in_n]_{C_j}} cost(in) = gain(I)$$

which contradicts the maximality of $gain(I)$.

Hence, $[in_1, \ldots, in_n]_{C_{j_1}} \in ValidOrders(C_{j_1})$ has a maximal cumulative cost in C_{j_1}. So, according to the definition of the gain (§ 3.3), we have for any overlapping component C_{j_1}:

$$\sum_{in \in [in_1, \ldots, in_n]_{C_{j_1}}} cost(in) = gain(C_{j_1})$$

Therefore, we have the result:

$$gain(I) = \sum_{1 \leq j \leq c} \sum_{in \in [in_1, \ldots, in_n]_{C_j}} cost(in) = \sum_{1 \leq j \leq c} gain(C_j)$$

□

B.3 Genetic Search

In this section, we prove desirable properties satisfied for each generation by roofers (Theorem 4) and misers (Theorem 5) during the genetic search (Section 8). We start by roofers.

Theorem 4 (Invariant of the Roofers). *For every generation n, we have:*

$$card(Roofers(n)) = n_{size}$$

$$\min\{cost(I) \mid I \in Roofers(n)\} = \min\{cost(I) \mid I \in \bigcup_{0 \le m \le n} Roofers(m)\}$$

and for every $I \in Roofers(n)$, we have:

- *I is reduced (in the sense of § 3.3)*
- *$Coverage(I) = Coverage_{obj}(C)$*

Proof. There are n_{size} individuals in the initial roofer population (§ 8.2). Moreover, the procedure detailed above to update the roofer population ensures that an offspring can only take the place of an existing roofer. Hence, the number of roofers does not change over generations.

In the initial population, a roofer I is always replaced by its reduced counterpart $reduc(I)$ after adding an input (§ 8.2). Moreover, after mutation (§ 8.5) each offspring I is replaced by its reduced counterpart $reduc(I)$ before determining if it is accepted in the roofer population or rejected. Hence, for each generation, each roofer is reduced.

Roofers in the initial population are built so that they cover all the objectives (§ 8.2). Moreover, in the above procedure, an offspring can be added to the roofer population only if it covers all the objectives. Hence, for each generation, each roofer covers all the objectives.

Finally, in the above procedure one can remove an individual from the roofer population only if a less or equally costly roofer is found. Hence, the minimal cost amongst the roofers can only remain the same or decrease over generations. Therefore, the minimal cost in the last generation is the minimal cost encountered so far during the search. □

To ease the proof for misers (Theorem 5), we first prove in Lemma 16 that a miser I_0 can be removed between generation n and generation $n+1$ only by a dominating miser I. Then, we prove in Corollary 3 that being dominated is carried from generation to generation.

Lemma 15 (Transitivity of Pareto Dominance). *The $\succ$ relation (§ 3.4) is transitive i.e., for every input sets I_1, I_2, I_3, if $I_1 \succ I_2$ and $I_2 \succ I_3$, then $I_1 \succ I_3$.*

Proof. For each $0 \le i \le n$, we have $f_i(I_1) \le f_i(I_2)$ and $f_i(I_2) \le f_i(I_3)$, so $f_i(I_1) \le f_i(I_3)$. Moreover, there exists $0 \le i_1 \le n$ such that $f_{i_1}(I_1) < f_{i_1}(I_2) \le f_{i_1}(I_3)$ and there exists $0 \le i_2 \le n$ such that $f_{i_2}(I_1) \le f_{i_2}(I_2) < f_{i_2}(I_3)$. i_1 and i_2 can be the same or distinct. In any case, we have $f_{i_1}(I_1) < f_{i_1}(I_3)$ and $f_{i_2}(I_1) < f_{i_2}(I_3)$. □

Lemma 16. *For every generation n, if $I_0 \in Misers(n) \backslash Misers(n+1)$, then there exists $I \in Misers(n+1)$ such that $I \succ I_0$, where $\succ$ is the domination relation from § 3.4.*

Proof. In the miser population update (§ 8.6), I_0 can be removed from the miser population only if there exists a miser candidate I_1 such that $I_1 \succ I_0$. If I_1 is accepted in the population then $I = I_1$ satisfies the lemma.

Otherwise, I_1 is rejected only because there exists a miser I_2 such that $I_2 \succ I_1$. Hence, according to Lemma 15, we have by transitivity that $I_2 \succ I_0$. Note that there is at most two miser candidates per generation. If either I_1 is the only miser candidate or there exists a second miser candidate I_3 which does not dominate I_2, then I_2 is present in the next generation and $I = I_2$ satisfies the lemma.

Otherwise, there exists a second candidate I_3 such that $I_3 \succ I_2$. Hence, according to Lemma 15, we have by transitivity that $I_3 \succ I_0$. If I_3 is accepted in the population then $I = I_3$ satisfies the lemma.

Otherwise, I_3 is rejected only because there exists a miser I_4 such that $I_4 \succ I_3$. Hence, according to Lemma 15, we have by transitivity that $I_4 \succ I_0$. Because there is at most two miser candidates per generation, I_4 cannot be removed by another candidate. Therefore, I_4 is present in the next generation and $I = I_4$ satisfies the lemma. □

Corollary 3 (Dominance across Generations)**.** *For every generation n and for every $I_1 \in Misers(n)$, if there exists $0 \leq m \leq n$ and $I_2 \in Misers(m)$ such that $I_2 \succ I_1$, then there exists $I_3 \in Misers(n)$ such that $I_3 \succ I_1$.*

Proof. The proof is done by induction on $n - m$.

If $I_2 \in Misers(n)$ then $I_2 = I_3$ satisfies the corollary.

Otherwise, there exists a generation $m < g \leq n$ where I_2 was removed. In that case, according to Lemma 16, there exists $I_3 \in Misers(g)$ such that $I_3 \succ I_2$. Hence, according to Lemma 15, we have by transitivity that $I_3 \succ I_1$. Finally, because $n - g < n - m$, we have the result using the induction hypothesis on I_3. □

We finally prove, as expected, the properties satisfied by misers on each generation.

Theorem 5 (Invariant of the Misers)**.** *For every generation n, for every $I_1 \in Misers(n)$, we have:*

- *I_1 is reduced (in the sense of § 3.3)*
- *$Coverage(I) \subset Coverage_{obj}(C)$ (the inclusion is strict)*
- *There exists no $I_2 \in \bigcup_{0 \leq m \leq n} Misers(m)$ such that $I_2 \succ I_1$.*

Proof. The initial miser population is empty (§ 8.2), hence the properties trivially hold.

After mutation (§ 8.5) each offspring I is replaced by its reduced counterpart $reduc(I)$ before determining if it is accepted in the miser population or rejected. Hence, for each generation, each miser is reduced.

In the miser population update (§ 8.6), an offspring can be a candidate to the miser population only if it does not cover all the objectives.

Finally, the last property is proved for a miser $I_1 \in Misers(n)$. Let n' be the first generation when I_1 was accepted, so we have $n' \leq n$ and $I_1 \in Misers(n')$.

The proof is done by contradiction, assuming that there existed a generation $0 \leq m \leq n$ and a miser $I_2 \in Misers(m)$ such that $I_2 \succ I_1$. Let m' be the first generation when I_2 was accepted, so we have $m' \leq m$ and $I_2 \in Misers(m')$. We consider two cases.

If $m' \leq n'$ then, according to Corollary 3, there exists $I_3 \in Misers(n')$ such that $I_3 \succ I_1$. This contradicts the above procedure, because if I_1 was dominated it would not have been accepted in $Misers(n')$.

If $m' > n'$ then, because $I_2 \succ I_1$, according to the above procedure I_1 is removed so $I_1 \notin Misers(m')$. But $m' \leq m \leq n$ and $I_1 \in Misers(n)$, so I_1 was added between m' and n. Let g be the first generation when this occurred.

In that case we have $I_1 \in Misers(g)$, $0 \leq m' \leq g$, $I_2 \in Misers(m')$, and $I_2 \succ I_1$. So, according to Corollary 3, there exists $I_3 \in Misers(g)$ such that $I_3 \succ I_1$, which contradicts the fact that I_1 was accepted in $Misers(g)$. □