Achieving Last-Mile Functional Coverage in Testing Chip Design Software Implementations

Ming Yan[†], Junjie Chen[†*‖], Hangyu Mao[‡‖], Jiajun Jiang[†], Jianye Hao[‡], Xingjian Li[†], Zhao Tian[†], Zhichao Chen[†]
Dong Li[‡], Zhangkong Xian[§], Yanwei Guo[§], Wulong Liu[‡], Bin Wang[‡], Yuefeng Sun[§], Yongshun Cui[§]
[†]College of Intelligence and Computing, Tianjin University, Tianjin, China
[‡]Noah's Ark Lab, Huawei, Beijing, China [§]Hisilicon, Huawei, Beijing, China
[†]{yanming, junjiechen, jiangjiajun, xingjianli, tianzhao, chenzhichao99}@tju.edu.cn
[‡]{maohangyu1, haojianye, lidong106, liuwulong, wangbin158, sunyuefeng1}@huawei.com
[§]{xianzhangkong, guoyanwei2, cuiyongshun}@hisilicon.com

***Abstract*—Defective chips may cause huge losses (even disasters), and thus ensuring the reliability of chips is fundamentally important. To ensure the functional correctness of chips, adequate testing is essential on the chip design implementation (CDI), which is the software implementation of the chip under design in hardware description languages, before putting on fabrication. Over the years, while some techniques targeting CDI functional testing have been proposed, there are still a number of hard-to-cover functionality points due to huge input space and complex constraints among variables in a test input. We call the coverage of these points *last-mile functional coverage*.**

Here, we propose the first technique targeting the significant challenge of improving last-mile functional coverage in CDI functional testing, called LMT, which does not rely on domain knowledge and CDI internal information. LMT first identifies the relevant variables in test inputs to the coverage of last-mile functionality points inspired by the idea of feature selection in machine learning, so as to largely reduce the search space. It then incorporates Generative Adversarial Network (GAN) to learn to generate valid test inputs (that satisfy complex constraints among variables) with a larger possibility. We conducted a practical study on two industrial CDIs in Huawei to evaluate LMT. The results show that LMT achieves at least 49.27% and 75.09% higher last-mile functional coverage than the state-of-the-art CDI test input generation techniques under the same number of test inputs, and saves at least 94.24% and 84.45% testing time to achieve the same functional coverage.

***Index Terms*—Chip Design Testing, Test Generation, Functional Coverage, Machine Learning**

I. Introduction

In the present age, chips are one of the most important infrastructures, which enable the rapid development of new technology, e.g., artificial intelligence [1] and 5G connectivity [2]. Undoubtedly, reliable chips are very crucial; otherwise grave losses of life and property may be caused [3]–[5]. For example, a bug in Intel's Pentium processors made the computer miscalculate long division and thus cost Intel hundreds of millions of dollars in recalling defective chips [3]. To ensure the quality of chips, one of the most critical tasks is to ensure that the design of a chip conforms to the specification before expensive fabrication in industry. Specifically, developers first implement the chip design into a *software* program in a hardware description language, called *chip design implementation* (abbreviated as *CDI* in this paper). Then, testers conduct CDI functional testing to check whether all functionalities are as expected according to the specification [6]–[9]. That is, CDI functional testing is quite critical for chip quality assurance.

In CDI functional testing, a test input aims to simulate a user request during the chip use and can trigger some functionalities (e.g., the switch of 5G connectivity is turned on) of the CDI under test. A test input requires to set the values for a set of variables that are responsible to control simulated user requests. In this task, functional coverage is the most concerned metric in practice, as it is essential to test all the functionalities of a CDI before fabrication [8], [10]–[12]. Here, we call the functionalities to be covered *functionality points*, a group of functionality points sharing common properties *functionality group*, and the percentage of covered functionality points over all functionality points *functional coverage*. Both functionality points and functionality groups are declared in CDIs by developers [8]. That is, the core goal of CDI functional testing is to achieve high (ideally full) functional coverage. However, with the rapid increase of CDI complexity, achieving high functional coverage becomes much harder. As shown in the existing study [13], over 70% of time and resources in chip design are spent on CDI functional testing. Therefore, achieving high functional coverage efficiently can shorten the process of CDI functional testing and reduce the cost of chip design significantly.

Over the years, many test input generation techniques targeting CDI functional testing have been proposed, e.g., search-based [14]–[18] and deep-learning (DL) based techniques [19]–[22] (introduced in Section VI). Although they can achieve relatively high functional coverage within acceptable time, there are still a number of hard-to-cover functionality points (called *last-mile functionality points*). As demonstrated by our motivating study in Section II-C, the first 90% functional coverage of an industrial CDI can be achieved by running less than 1,000 randomly generated test inputs, but the last

*Junjie Chen is the corresponding author.
‖The two authors contributed equally to this work.

10% functional coverage requires to continue running more than 49,000 test inputs. Achieving the last-mile functionality points often requires experts to manually design test inputs, which usually takes a very long time, even more than half of the entire chip-design time [13]. The reasons for the significant challenge of covering last-mile functionality points are threefold:

- There are a large number of variables to be set in a test input, many of which have a wide value range, leading to enormous input space. For example, the input of a CDI used in our study contains 575 variables, constituting the input space with over 10^{1025} settings.
- There are vast and complex constraints among variables [23], especially covering last-mile functionality points involves much more strict constraints. For example, the constraints of a CDI used in our study are defined by more than 10K lines of source code and each constraint often involves dozens of variables (with dependency).
- CDI functional testing prefers nearly black-box methods in practice [7]. This is because CDI development teams and functional testing teams are often independent of each other, whose reasons include: (1) CDIs often have high security levels in industry; (2) it can facilitate to test CDIs as required rather than as programmed; (3) it can improve the generality of testing methods and avoid high cost.

Overall, identifying test inputs satisfying those strict constraints from the huge input space in a nearly black-box way is definitely difficult. Although there are some test input generation techniques proposed for traditional software, they are not applicable to CDI functional testing due to the following reasons: (1) the scale of CDI input space is significantly large (in industrial grade), which hinders the application of many techniques (such as combinatorial testing [24]–[27]); (2) many testing techniques are white-box (depending on program internal information), e.g., symbolic execution [28]–[31] and source-code-based fuzzing techniques [32]–[34]; (3) there are few tools supporting the analysis of code written in hardware description languages, which hinders the application of testing techniques proposed/implemented for the software programmed in high-level programming languages. More discussions on traditional software testing techniques will be presented in Section VI. Hence, achieving last-mile functional coverage is still an *open challenge* in CDI functional testing.

In this work, we target the bottleneck problem in CDI functional testing, i.e., *achieving last-mile functional coverage*, so as to save experts' efforts and speed up the whole CDI functional testing process. Specifically, we propose the first technique targeting last-mile functional coverage, called **LMT** (**L**ast-**M**ile **T**est generation), which does not rely on domain knowledge and CDI internal information. Its key insight is to reduce the search space for test inputs, and then identify desirable test inputs (that have to satisfy complex constraints) in the reduced space for efficiently covering last-mile functionality points. Specifically, LMT identifies a small portion of most relevant variables for each last-mile functionality group by formulating this problem as the problem of feature selection in machine learning (ML) [35]–[37]. It is more likely to cover the functionality points in the functionality group by sufficiently exploring the settings of these relevant variables. To overcome the challenge of satisfying complex constraints among variables, LMT incorporates Generative Adversarial Network (GAN) [38] to learn constraints so as to construct valid test inputs corresponding to the given settings of relevant variables and thus make them indeed take effect. In this way, LMT can generate effective and valid test inputs targeting the last-mile functionality points efficiently, and thus largely improve last-mile functional coverage.

To evaluate the effectiveness of LMT, we conducted a practical study based on two industrial CDIs in an international company (i.e., Huawei) that develops and designs a series of widely-known chips. Due to the confidentiality policy of Huawei, we hide the names of both CDIs. The experimental results demonstrate the superiority of LMT over the state-of-the-art test input generation techniques in terms of last-mile functional coverage. Specifically, LMT covers at least 49.27% and 75.09% more last-mile functionality points than the state-of-the-art techniques under the same number of test inputs on the two CDIs, respectively. Also, LMT saves at least 94.24% and 84.45% testing time to achieve the same functional coverage as the state-of-the-art techniques on the two CDIs, respectively. In particular, the effectiveness of LMT has been largely appreciated by Huawei and LMT has been deployed on six CDIs in Huawei as the standard testing technique (replacing the previous practice).

This work makes the following major contributions:

- We propose the first technique targeting last-mile functional coverage in CDI functional testing, called LMT, which does not rely on domain knowledge and CDI internal information.
- We model the relevance between variables and a targeted last-mile functionality group from the view of feature selection, in order to identify the relevant variables for covering the last-mile functionality points in this group.
- We adopt GAN to learn constraints among variables involved in a test input, in order to have a larger possibility to generate valid test inputs.
- We conducted a practical study on two industrial CDIs, demonstrating the effectiveness of LMT. In particular, LMT has been deployed in Huawei for practical use.

Note: The targeted problem of this work is a critical problem in chip companies and definitely a software-engineering task (as CDIs are also a kind of software). In fact, we are the first to bring it to the software-engineering community, and formulate this problem and its unique challenges from the software-engineering view. We hope to attract more attention and wisdom from this creative community to further solve this challenging problem.

II. Background and Motivation

A. Chip Design Implementation

Unlike traditional software that can be continuously developed and improved even though being deployed, the develop-

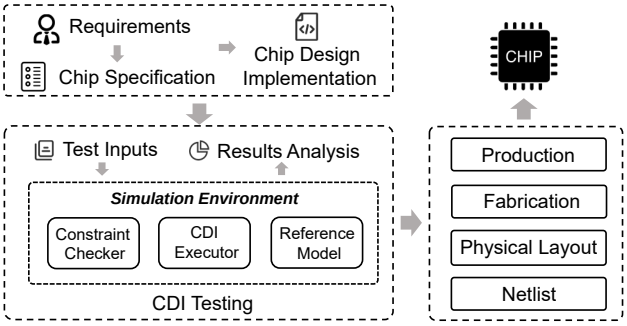


Fig. 1: The general process of chip development

ment of chips is a one-off activity. Once designed chips are put on fabrication, they cannot be further modified. The fabrication is expensive, and thus defective chips can cause huge economic losses. Hence, to ensure the reliability of chips, the testing of designed chips before fabrication is very critical. Figure 1 shows the general process of chip development. When providing the chip requirements, chip developers first construct the formal specification and then implement it into a *software program* in a hardware description language (e.g., SystemVerilog), called *chip design implementation* (CDI). Please note that CDI realizes the same functionalities as the hardware chip under design according to the given specification.

As the CDI is a program, it can be run and tested like traditional software. The difference is that it runs in a particularly designed environment that simulates the usage conditions of the hardware chip in practice (to be presented in Section II-B). After sufficient testing, a CDI is synthesized into Netlist, a machine-readable representation that contains corresponding electronic components and their connections. Finally, the physical layout of components is generated according to the Netlist, which is ready for hardware production.

B. CDI Testing

As shown in Figure 1, the general CDI testing process mainly contains three steps: test input generation, test execution by a simulator, and test result analysis. When providing a valid test input, the simulator can execute the CDI and report the execution results. Note that functionality points are declared in the CDI by developers and thus the simulator (e.g., Synopsys VCS) can check which functionality points are covered by a test input. Due to the complex chip functionalities, there are vast and complex constraints among variables. The test inputs violating the constraints are regarded as invalid. To make the test inputs safe and meaningful, a builtin constraint checker of the simulator is responsible to check the validity of test inputs and can change invalid inputs to be valid via simple rule-based modifications.

Figure 2 shows a simplified code snippet of constraints from an industrial CDI used in our study, which is described in SystemVerilog (which is a popular hardware description language). We simplified the constraint for ease of illustration, but in fact the constraints often involve dozens of variables (with dependency). Here, `mode` and `flag` are two variables in the test input. The constraints require that when `flag` equals

```
1 if (flag==0)\
2     mode inside {4,5,6}; \
3 else \
4     mode inside {1,2,3};
```

Fig. 2: An example of variable constraints in CDI

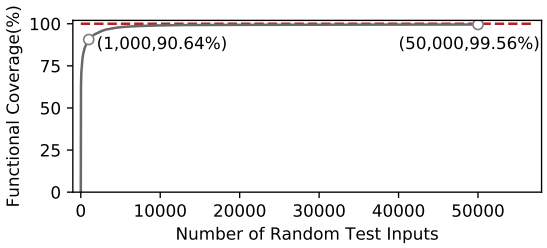


Fig. 3: Functional coverage by random test generation

to 0, `mode` should be set in {4, 5, 6}; otherwise, in {1, 2, 3}. Given t=(`flag`, `mode`) as the simplified input format of the CDI, the input t_0=(1, 3) is regarded as valid while t_1=(0, 1) as invalid. Then, the constraint checker modifies t_1 to a valid input t_1'=(0, 4) according to the embedded rules before feeding to the CDI for execution. Note that even though the embedded rules can help make an invalid test input to be valid, they can also limit the modified inputs to cover a fixed and limited set of functionality points and thus negatively affect their testing capability in improving functional coverage.

As CDI testing is responsible for the reliability of chips, ensuring the adequacy of CDI testing is vitally important. In traditional software testing, code coverage (e.g., statement coverage) is widely adopted to measure test adequacy from the perspective of code structure. However, due to the complex chip functionalities, even though all statements have been covered by test inputs, some functionalities (corresponding to particular input settings) may be still untested, and possibly perform improperly. For example, if a certain functionality is missed to implement in the CDI, structural coverage cannot discover it while functional coverage (that actually measures the coverage of design intent from the perspective of functional requirements) can. For CDI testing, it is essential to test all chip functionalities [39]. Hence, *functional coverage* is widely-used in practice [8], [10]–[12]. Here, we call the CDI testing that aims to improve functional coverage *CDI functional testing*. The key concepts (e.g., functionality points and functionality groups) in CDI functional testing have been introduced in Section I. Our work also focuses on CDI functional testing.

Note that chip developers manually construct a reference model for checking the output correctness of each test input on the CDI. It is also important to improve the test checking process, but it is out of the scope of our work. Similar to most of CDI functional testing work [10], [22], [40], our work aims to improve functional coverage of the CDI under test, but different from them, our work targets the open challenge of improving last-mile functional coverage (Section II-C).

C. Last-Mile Functional Coverage

Due to the huge input space and complex constraints among input variables, there are usually a number of functionality

points that are hard to be covered, even with the state-of-the-art test input generation techniques (also confirmed in Section IV). To clearly understand this challenging problem, we conducted an experiment on an industrial CDI that includes over 7,000 functionality points (called $\mathcal{MA}$ to be introduced in Section IV). Specifically, we adopted the random test input generation technique that is widely used in industry, to generate test inputs, and then analyzed the achieved functional coverage.

Figure 3 shows the results, where the *x-axis* represents the number of generated test inputs and the *y-axis* represents the achieved functional coverage. From this figure, the first 1,000 test inputs have achieved 90.64% functional coverage, while the subsequent 49,000 test inputs increase only 8.92% and still cannot achieve full functional coverage in the end. That is, during CDI functional testing, most of the time is spent on achieving the last 10% functional coverage. We call those hard-to-cover functionality points *last-mile functionality points*. In practice, last-mile functionality points could be determined by observing the growth curve of functionality points under randomly generated test inputs. When the curve becomes closely flat, the uncovered functionality points can be regarded as last-mile functionality points.

As shown in our study (Section IV), even the most advanced DL-based test input generation technique still suffers from the unsatisfactory performance facing the last-mile functionality points. In industry, extensive experts' manual efforts have to be devoted to designing effective test inputs for covering them, which is labor-intensive and time-consuming. Hence, an effective and efficient test input generation technique targeting the last-mile functionality points is urgently desired.

III. Approach

We propose the first technique targeting *last-mile functional coverage* in CDI functional testing, called **LMT** (**L**ast-**M**ile **T**est generation), which aims to efficiently achieve last-mile functional coverage for saving experts' manual efforts.

To overcome the enormous search space of test inputs, LMT learns the relevance between input variables and each targeted functionality group so as to identify the most relevant variables to the group. In this way, testing can be more targeted by focusing on exploring the settings for a small set of relevant variables rather than all the variables, and thus the last-mile functionality points in the group can be covered more efficiently. In particular, we transform the problem of relevant variable identification as the problem of feature selection in ML, and solve it by building a ML model (i.e., Random Forest [41] in LMT). To make the explored settings take effect to cover the targeted functionality points, LMT needs to set the remaining less relevant variables to make the generated input as valid as possible. Here, LMT adopts a Generative Adversarial Network (GAN) model to learn constraints among variables to complete the task. Finally, we design a heuristic-based strategy to combine the two components and guide the whole test input generation process, so as to achieve the last-mile functional coverage efficiently. Figure 4 shows the overview of LMT.

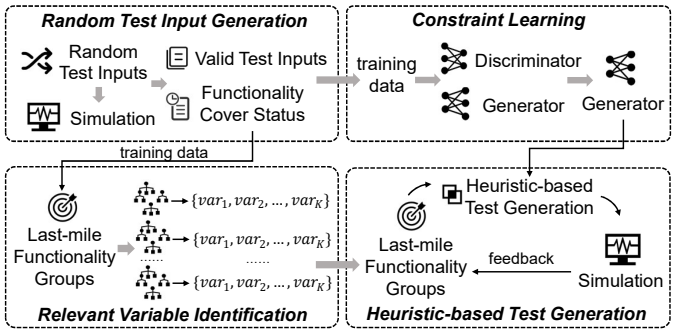


Fig. 4: Overview of LMT

Please note that our work targets the coverage of last-mile functionality points, and thus we first leverage the lightweight random test input generation technique to generate a set of test inputs for covering those easy-to-cover functionality points, which is effective and efficient as shown in our study (Section II-C) and also widely used in industry. Then, LMT dedicates to those hard-to-cover functionality points (i.e., last-mile functionality points), and takes these randomly generated inputs as training data for building the ML and GAN models.

A. Relevant Variable Identification

As explained above, to overcome the enormous input space and make the generated test inputs more targeted, i.e., covering particular functionality points, LMT identifies the most relevant variables for each last-mile functionality group (i.e., the group containing the last-mile functionality points). The intention of relevant variable identification is actually aligned with the idea of feature selection in ML, which aims to identify the features that make significant contributions to a given prediction task. Here, we adopt the Random Forest (RF) method to model the relevance due to the following reasons: 1) *High interpretability*. Input features in the RF model are used to build branch conditions in decision trees, where the importance of features can be well explained as the ability of discrimination; 2) *Great performance*. It has been widely used in feature selection tasks due to its well-recognized effectiveness and high efficiency [42]–[45], and it is also scalable to the large number of input variables in our task. Please note that LMT is not specific to RF and can be integrated with other feature selection methods. In particular, we have conducted a preliminary experiment to evaluate LMT under some other typical feature selection methods (i.e., XGBoost [46] and Low Variance Filter [47]), and the results confirm the superiority of RF in our scenario. In the future, we will evaluate LMT under more advanced feature selection methods (such as LASSO [37]). Next, we introduce the relevance model building and relevant variable identification process in detail.

Formally, we define $\mathcal{G} = \{g_1, g_2, \ldots, g_n\}$ as the set of last-mile functionality groups of the CDI under test, where $g_i = \{p_i^1, p_i^2, \ldots, p_i^{m_i}\}$ denotes the i^{th} functionality group that consists of m_i functionality points. Different functionality groups may include different number of functionality points, and $g_i \cap g_j = \emptyset$ if $i \neq j$. Besides, we use $\mathcal{V} = \{v_1, v_2, \ldots, v_d\}$

to represent the set of input variables of the CDI, and d is the input length. Therefore, the target of the relevant variable identification process in LMT is to pinpoint a subset of variables $V_{g_i} \subsetneq \mathcal{V}$ for each $g_i \in \mathcal{G}$.

To build the relevance model via RF, LMT requires training data. As mentioned before, it uses the randomly generated inputs for covering easy-to-cover functionality points, denoted as $\mathcal{T} = \{t_1, t_2, \ldots, t_u\}$ (each t_i is composed by a set of values corresponding to the variables in $\mathcal{V}$), to construct training data, and the functional coverage for the target functionality group (denoted as g_i) achieved by each test input as the corresponding label. The functional coverage can be automatically obtained by feeding the test input to the simulator for execution. As our work targets last-mile functional coverage, it is essential to execute these randomly generated inputs for the entire CDI functional testing process. Thus, obtaining the training data for our relevance model is natural without extra cost (especially manual effort). As more than one functionality points in g_i may be covered by a single test input, the functional coverage for t_i may be a set of functionality points $\mathcal{P}_{t_i} \subseteq g_i$. Specifically, we denoted the collected training set as $\mathcal{D} = \{(t_1, l_{t_1}), (t_2, l_{t_2}), \ldots, (t_u, l_{t_u})\}$, where $l_{t_i} = \langle l^1_{t_i}, l^2_{t_i}, \ldots, l^{m_i}_{t_i} \rangle$ is a vector of 0 and 1 *s.t.* $\forall 1 \leq j \leq m_i, l^j_{t_i} = 1$ *iff* $p^j_i \in \mathcal{P}_{t_i}$ (m_i is the size of g_i). Then, a relevance model can be trained via RF over $\mathcal{D}$ by taking t_i as the feature vector and l_{t_i} as the label. Please note that since m_i tends to be larger than 1, the built model is usually a multi-label classification model.

After building the relevance model for a targeted group g_i, LMT then measures the most relevant variables to g_i. Inspired by the feature selection process, we adopt the Gini importance to measure the relevance of each variable to g_i, which is widely-used and has been demonstrated to be effective [44], [45]. Specifically, each internal node τ in a decision tree splits a set of data $\mathcal{D}_\tau$ into different classes by a certain condition over some input feature. The Gini importance of the node reflects the total reduction of Gini impurity, which measures the likelihood of misclassifying a randomly chosen data from $\mathcal{D}_\tau$. That is, Gini importance can be viewed as the ability of features taken by the node to correctly identify different classes. In our scenario, it represents the ability of a variable for discriminating different functionality points in the group. More details on the calculation of Gini importance can refer to the original theory [41], [43].

Then, all the variables can be ranked in the descending order based on their relevance scores, and Top-*K* variables are identified by LMT as the most relevant ones to g_i. In this way, LMT can identify the K most relevant variables with regard to each last-mile functionality group. Subsequently, to achieve the last-mile functional coverage, LMT focuses on exploring the settings of the K variables relevant to each last-mile functionality group, instead of the settings of all the variables in a test input, which can be more targeted and largely reduce the search space.

B. Constraint Learning

A valid test input has to meet complex constraints as presented in Section II, to make the settings of the *K* variables actually take effect, the settings of the remaining variables are also critical and thus should be carefully determined. Facing this challenge, LMT adopts GAN to learn to generate test inputs satisfying complex constraints among variables with a larger possibility. GAN has been well-recognized and widely adopted with the goal of generating new data that satisfy the given distribution [48], [49], which is aligned with our purpose that makes generated inputs as valid as possible by learning from known valid inputs.

A typical GAN model consists of a generator and a discriminator. The former aims to learn the statistical distribution information from the given training data and takes the responsibility for data generation, whereas the latter aims to correctly distinguish the newly generated data from the training data. Both the generator and the discriminator are neural networks composed of several fully-connected or/and convolutional layers. We use $\mathcal{M}_g$ and $\mathcal{M}_d$ to denote the generator and discriminator models, respectively. During the training process, the two models combat with each other until reaching a relatively balanced state according to the given target, i.e., the loss function. Then, the built generator $\mathcal{M}_g$ can be used to generate new data that have a larger possibility to share the same distribution with training data.

In our scenario, the GAN model is trained by feeding a set of valid test inputs. Similar to the training set for building the RF model, we also take the test inputs that are randomly generated but modified by the constraint checker (if invalid) as the training data $\mathcal{D}$. In this way, the generator is expected to learn what a valid input looks like by *implicitly* learning the complex constraints among variables based on $\mathcal{D}$. Specifically, during the training process, the model $\mathcal{M}_g$ in LMT generates a set of candidate test inputs $\mathcal{M}_g(\boldsymbol{z})$ for the CDI when taking $\boldsymbol{z}$ as input, where $\boldsymbol{z}$ is randomly generated according to the given prior distribution $p_z(\boldsymbol{z})$. Then, newly generated inputs are fed to $\mathcal{M}_d$ along with a set of samples $\boldsymbol{x}$ from the training data $\mathcal{D}$ (i.e., $\boldsymbol{x} \sim p_{\mathcal{D}}(\boldsymbol{x})$). Finally, the model is updated according to the loss function in Formula 1 via back-propagation.

$$\begin{aligned} \min_{\mathcal{M}_g} \max_{\mathcal{M}_d} \mathcal{L}(\mathcal{M}_d, \mathcal{M}_g) = \mathbb{E}_{\boldsymbol{x} \sim p_{\mathcal{D}}(\boldsymbol{x})}[\log \mathcal{M}_d(\boldsymbol{x})] \\ + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - \mathcal{M}_d(\mathcal{M}_g(\boldsymbol{z})))] \end{aligned} \tag{1}$$

where $\mathcal{M}_d(\cdot)$ and $\mathcal{M}_g(\cdot)$ denote the outputs of corresponding models, while $p_{\mathcal{D}}$ and p_z represent the distribution of training data $\mathcal{D}$ (i.e., known valid test inputs) and the prior distribution for $\mathcal{M}_g$ (i.e., the normal distribution [50]) respectively. Specifically, the loss function means that when given $\mathcal{M}_g$, we first train the model $\mathcal{M}_d$ via *maximizing* the value of $\mathcal{L}(\cdot)$ to make it correctly discriminate generated data from training data. Then, given $\mathcal{M}_d$, we train the model $\mathcal{M}_g$ by *minimizing* the value of $\mathcal{L}(\cdot)$, which makes $\mathcal{M}_g$ have the ability to generate test inputs under the distribution close to $\mathcal{D}$. That is, the generated test inputs are more likely to be valid. Note that the GAN

Algorithm 1: Heuristic-based Test Input Generation

```
Input : M: The number of settings for the K relevant variables
        N: The number of test inputs generated by GAN
        Φ: Pre-defined maximum number of generated test inputs
        G: The list of last-mile functionality groups
        RV: The dictionary recording relevant variables.
Output: TS: The generated test inputs by LMT.
1  G ← sort(G) /* Descending order of uncovered
       functionality points.                                  */
2  TS ← emptySet()
3  i ← 1
4  while Size(TS) < Φ do
5      V_{G[i]} ← RV[G[i]]/* Get K most relevant variables    */
6      R ← rvSampling(V_{G[i]}, M) /* Produce M settings for
           V_{G[i]}                                           */
7      S_0 ← candidateGeneration(gan, N) /* Generate N candidate
           test inputs through the GAN model gan              */
8      S_1 ← candidateSelection(R, S_0) /* Select the most similar
           test input with S_0 for each setting in R          */
9      S ← testInputsComposition(S_1, R) /* Combine S_1 and R  */
10     TS ← TS ∪ S
11     Cov ← simulation(S, CDI)/* Simulate S and collect
           coverage                                           */
12     updateCoverage(Cov) /* Update coverage groups in G     */
13     if achieveFullCoverage() then
14         break /* Covering all functionality points        */
15     end
16     i ← i + 1
17     if i > Size(G) then
18         G ← removeCoveredGroup(G) /* Remove the
               functionality groups without uncovered
               functionality points from G                    */
19         G ← sort(G)
20         i ← 1
21     end
22 end
23 return TS
```

model does not output specific constraints, but *implicitly* learns constraints to *generate valid inputs with a large possibility*.

Overall, when providing the settings of the K relevant variables (identified by the RF model), the remaining variables can be properly set with the guide of the GAN model. To increase the possibility of producing a valid and effective test input, we design a novel heuristic-based strategy in LMT to guide the process of test input generation by combining the superiority of both RF and GAN models (in Section III-C).

C. Heuristic-based Test Input Generation

Based on the RF and GAN models, we design a heuristic-based strategy of test input generation for CDI functional testing, so that those last-mile functionality points can be covered as *efficiently* as possible. LMT is actually an iterative process, which first selects a targeted last-mile functionality group and then generates test inputs based on the RF and GAN models with the purpose of covering more uncovered functionality points in this group in each iteration. Algorithm 1 formally illustrates the iterative test generation process in LMT.

Regarding the selection of a targeted last-mile functionality group, LMT selects the functionality group with the largest number of uncovered functionality points as the targeted one iteratively (Line1 and Line19). This is because it can be more beneficial to improve the overall last-mile functional coverage by first generating test inputs targeting such a group. After selecting a targeted functionality group, LMT uses the corresponding RF model to identify the K relevant variables to this group (Line5). Sufficiently exploring the settings of the K relevant variables could facilitate to improve functional coverage in this group. Here, LMT randomly explores M settings for the K relevant variables (Line6), as randomness has the inherent characteristic to keep diversity. Indeed, we can incorporate other more advanced methods (than random exploration) in this step, which will be our future work.

To make an explored setting of the K relevant variables take effect for facilitating the functional coverage of the targeted functionality group, it is also required to properly set the remaining variables so as to produce a valid test input. Here, LMT uses the GAN model to generate a set of test inputs to guide the proper setting of remaining variables: First, LMT generates N candidate test inputs based on the GAN model (Line7); Then, for each explored setting of the relevant variables, it measures the similarity between the explored setting and the setting on those K relevant variables in each candidate test input via the Euclidean metric as shown in $Dist_{X,Y} = \sqrt{\sum_{v \in V_{g_i}} (X[v] - Y[v])^2}$, where X denotes a setting of the K relevant variables (i.e., V_{g_i}) for the group g_i, whereas Y denotes a setting of all the variables in a candidate test input generated by the GAN model. In this way, when providing the setting X, the most similar setting Y can be identified among those N candidates generated by the GAN model (Line8). Then, the settings of remaining variables (except the K relevant ones) in Y can be integrated with X to form a complete test input (Line9), which is more likely to be valid. After generating M test inputs in this way, they are simulated and functional coverage results are collected (Line11). Although the M test inputs are produced with the purpose of covering the last-mile functionality points in the targeted group, they may also cover some uncovered functionality points in other groups. Therefore, the coverage of each last-mile functionality group should be updated (Line12).

In particular, instead of existing valid test inputs in training data, using newly generated inputs via the GAN model for the settings of remaining variables can increase the diversity of test inputs, which is helpful to improve the overall functional coverage. We also conducted a small experiment to compare the two methods and our results confirmed the conclusion. Specifically, for a CDI (called $\mathcal{MA}$ introduced in Section IV), using existing valid inputs in training data only covers 24 functionality points among 69 last-mile functionality points after running 30,000 test inputs, while LMT covers 67 points.

If full functional coverage is achieved, the test input generation process terminates (Line13-15); Otherwise, the next functionality group is selected as the targeted one for the next iteration (Line16). After all the functionality groups have been visited once, LMT updates the set of last-mile functionality groups (by removing the functionality groups without uncovered functionality points from $\mathcal{G}$), and re-sorts the remaining last-mile functionality groups as the descending order of the updated number of uncovered functionality points in each functionality group (Line17-21). Here, we do not re-sort the last-mile functionality groups after each iteration, as it may lead to frequently selecting the same functionality group as the targeted one, which could damage the cost-effectiveness of

TABLE I: Basic information of CDIs under test

CDI	# Var	# FP	# FG	# SLOC
$\mathcal{MA}$	575	7,219	681	40K+
$\mathcal{MB}$	672	81,563	353	40K+

LMT from the perspective of improving the overall functional coverage. Besides achieving full functional coverage, the test input generation process also terminates when the pre-defined maximum number of generated test inputs is reached (Line4).

IV. Evaluation

In our study, we aim to address three research questions:

- **RQ1**: How does LMT perform in achieving last-mile functional coverage compared with state-of-the-art techniques?
- **RQ2**: Does each main component in LMT contribute to the overall effectiveness of LMT?
- **RQ3**: What is the influence of the number of identified relevant variables on the effectiveness of LMT?

A. Experimental Study Design

Subjects. We conducted our study on two industrial CDIs (named $\mathcal{MA}$ and $\mathcal{MB}$) in Huawei, which designs and develops a series of widely-known chips. Due to the company policy, we hide the CDI names. Both $\mathcal{MA}$ and $\mathcal{MB}$ are designed for network devices, which perform time-domain and frequency-domain processing (e.g., time and frequency offset estimation) in digital communication systems. Since they need to support various network protocols, the constraints among input variables in them are very complicated.

Table I shows the basic information of both CDIs, where the last four columns present the number of variables in the test input, the number of functionality points, the number of functionality groups, and the number of source lines of code of the CDI, respectively. Due to the company policy, we just provide the rough SLOC for the two CDIs. Indeed, both of them are large-scale. As suggested by the developers, we regard the remaining uncovered functionality points after running 10,000 (for $\mathcal{MA}$) and 15,000 (for $\mathcal{MB}$) test inputs via random test input generation, as the last-mile functionality points. This is because the functional coverage can hardly increase after running those test inputs (also confirmed in Figures 5a and 5b). As a result, there are 69 and 25,907 functionality points left for $\mathcal{MA}$ and $\mathcal{MB}$ respectively, which are regarded as the last-mile functionality points and thus the target of our technique.

Implementation and Configurations. We implemented the component of relevant variable identification based on the RF algorithm provided by *scikit-learn* [51]. We adopted the state-of-the-art GAN (i.e., WGAN-GP [52]) to implement the component of constraint learning, which is implemented based on PyTorch 1.4.0. Following the existing work [53], [54], the generator of GAN consists of four linear layers, whose input dimension is 100 and output dimension is the same as the number of variables for the CDI. The discriminator consists of five linear layers, whose input dimension is the same as the number of variables for the CDI and output dimension is 1. Regarding parameters in RF and GAN, we set them via grid search on a small dataset, e.g., setting the number of epochs to 20 in GAN and the number of trees to 50 in RF. All the settings are the same on both $\mathcal{MA}$ and $\mathcal{MB}$ and recommended as the default settings in LMT. The complete parameter settings can be found on our project homepage [55].

By balancing the efficiency and effectiveness, we set the default setting of K (the number of identified relevant variables to a targeted functionality group) to 50 in LMT. Also, we investigated the influence of K on the effectiveness of LMT in RQ3 by setting K to 10, 30, 50, and 100, respectively. Moreover, we set M (the number of explored settings for identified relevant variables) to 1,000 and N (the number of candidate test inputs generated by the GAN model) to 5,000.

Measurements: LMT aims to achieve *higher* last-mile functional coverage *more efficiently*, which is indeed the critical goal of CDI functional testing in practice. Hence, we used two metrics to measure the effectiveness of LMT compared with the state-of-the-art test input generation techniques: 1) the improvement on last-mile functional coverage under the same number of test inputs, and 2) the reduction on CDI functional testing time when reaching the same functional coverage. Each studied technique generates 30,000 test inputs on the basis of the set of test inputs generated randomly for covering easy-to-cover functionality points, so as to sufficiently investigate its ability of achieving last-mile functional coverage. To reduce the influence of randomness, we repeated all the experiments 5 times, and reported the average results in the study. Our experiments were conducted on an Intel Xeon Gold 6278C machine with 512GB RAM, Centos 7.9.2009.

Please note that we include the time spent on training RF and GAN models into the testing time with LMT. The process of training data collection is exactly the process of executing randomly generated inputs for covering easy-to-cover points, and thus it does not incur extra cost.

B. RQ1: Overall Effectiveness of LMT

1) Setup: To evaluate the effectiveness of LMT, we compared LMT with three existing techniques, including one baseline and two adapted state-of-the-art techniques. The baseline is **random test input generation**, which randomly sets the value of each variable in a test input. The other two techniques are **GA-based** (Genetic-Algorithm-based) and **DL-based** test input generation techniques. As there is no existing technique specialized for achieving last-mile functional coverage, we adapted the widely-studied GA-based and DL-based test input generation techniques in our study to fit our scenario for sufficient comparison.

GA has been widely used in test input generation, including both CDI functional testing [14]–[18] and general software testing [56]–[59]. All these GA-based test input generation techniques share the common components, including a fitness function for guiding the test input generation process, and selection, crossover, and mutation operators for producing test inputs in each iteration. Inspired by existing GA-based techniques [14]–[18], the adapted GA-based technique in our

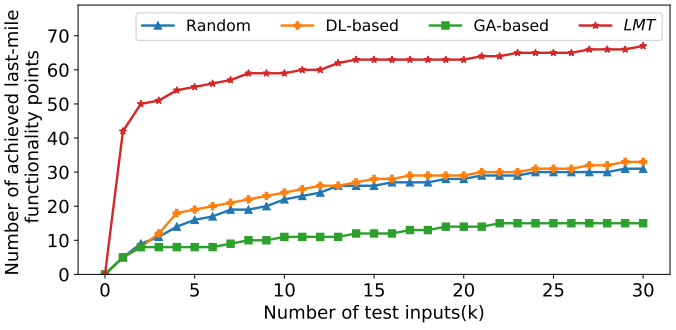
scenario takes a test input as an individual and the number of covered last-mile functionality points as the fitness function, which is aligned with our goal of improving last-mile functional coverage. The GA-based technique is implemented based on the public artifact [60]. We empirically set the ratio of *mutation* and *crossover* to 0.4 and 0.8 respectively (which is the best setting in our study based on a small dataset), while using the default settings for the other parameters.

Also, we adapted the state-of-the-art DL-based CDI test input generation technique [22] in our scenario by modeling the relationship between test inputs and functionality points. It takes the test inputs randomly generated for covering easy-to-cover points as training data and the functional coverage of a test input on each functionality point as the label of the input. The output of the model is the probability of each functionality point being covered by a generated test input. If all the functionality points with large predicted probabilities (over 0.6 in our study) for a generated test input have been covered before, it filters out this input as it is less likely to improve functional coverage. The newly generated inputs are also used to fine-tune the model for 10 epochs and the weights with the best performance are saved for subsequent iterations.

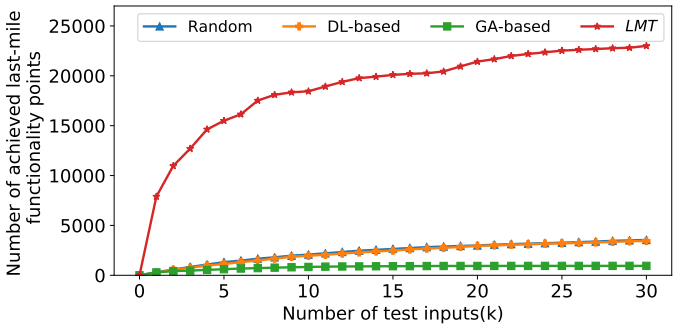
Intuitively, **combinatorial testing** (widely used in traditional software testing) may also fit our scenario, but we did not study it as the number of input variables in a CDI is very large and most of them have a large value range, leading to extremely huge space for combinatorial testing. Such huge space can lead to unaffordable time cost for combinatorial testing. Here, we conducted a small experiment by applying a widely-studied combinatorial testing tool (i.e., PICT [61]) to the first 25 input variables in $\mathcal{MA}$. This tool spent more than 11 hours for achieving the *2-way* combinatorial coverage. Obviously, if we apply it to all the input variables in our studied CDIs (over 500), the time cost is unaffordable.

2) Results and Analysis: Figure 5 shows the comparison results between LMT and the three compared techniques, where the *x-axis* represents the number of generated test inputs and the *y-axis* represents the number of achieved last-mile functionality points. From this figure, for the widely-used random test input generation in practice, after running all the 30,000 test inputs, there are still a large number of uncovered last-mile functionality points on both CDIs. Specifically, 55.07% (38 out of 69) last-mile functionality points are still uncovered on $\mathcal{MA}$ while 86.32% (22,362 out of 25,907) last-mile functionality points are still uncovered on $\mathcal{MB}$, further confirming the significant challenge of achieving last-mile functional coverage.

In Figure 5, LMT largely improves last-mile functional coverage on both $\mathcal{MA}$ and $\mathcal{MB}$. The line representing LMT is significantly higher than those representing the compared techniques in this figure. After running only 2,000 test inputs generated by LMT, 72.46% (50 out of 69) last-mile functionality points have been achieved on $\mathcal{MA}$. When running all the generated 30,000 test inputs, only 2.90% (2 out of 69) last-mile points are uncovered on $\mathcal{MA}$. On $\mathcal{MB}$ (a larger CDI with much more last-mile functionality points), after running all the generated 30,000 test inputs, 88.77% (22,998 out of 25,907) last-mile functionality points are achieved.



(a) $\mathcal{MA}$



(b) $\mathcal{MB}$

Fig. 5: Comparison between LMT and the existing techniques

In contrast, on $\mathcal{MA}$, the most effective compared technique is the DL-based technique, but it still largely performs worse than LMT and achieves only 47.83% last-mile functional coverage after running all the 30,000 test inputs. On $\mathcal{MB}$, all the three compared techniques have similar effectiveness, but largely perform worse than LMT, e.g., achieving at most 13.68% last-mile functional coverage after running all the 30,000 test inputs. **When running 30,000 generated test inputs, LMT achieves at least 49.27% and 75.09% higher last-mile functional coverage than all the compared techniques on $\mathcal{MA}$ and $\mathcal{MB}$, respectively.**

Besides, on $\mathcal{MA}$, the best compared technique (DL-based test input generation) spent 11.79 hours on running all the 30,000 test inputs for achieving 47.83% last-mile functional coverage, but LMT spent only 0.68 hours (that has included the time spent on building RF and GAN models, i.e., 0.35 hours) on achieving the same coverage. Similarly, on $\mathcal{MB}$, the best compared technique spent 12.51 hours on running 30,000 test inputs for achieving 13.68% last-mile functional coverage, but LMT spent only 1.95 hours (including 1.75 hours on building RF and GAN models) on achieving the same coverage. **Overall, LMT saves 94.24% and 84.45% CDI testing time compared with the most effective compared technique among the three on $\mathcal{MA}$ and $\mathcal{MB}$, respectively, demonstrating the significant superiority of LMT.** In particular, according to the results, even though LMT spends extra time cost on building RF and GAN models, it still outperforms the compared techniques in achieving last-mile functional coverage. This is because that its extra time cost is acceptable and significantly smaller than

TABLE II: Percentage of valid cases

CDI	*Random*	*DL*	*GA*	LMT$_{noGAN}$	LMT$_{noRF}$	LMT
$\mathcal{MA}$	33.72%	35.19%	39.40%	33.71%	63.58%	**65.05%**
$\mathcal{MB}$	51.94%	52.16%	53.39%	51.98%	84.45%	**86.30%**

the time cost spent on running test inputs.

In particular, LMT achieves full functional coverage on $\mathcal{MA}$ at two (out of five) times, while the compared techniques do not at all the five times. Due to the complexity of $\mathcal{MB}$, LMT does not achieve full functional coverage on it during the same testing time as $\mathcal{MA}$. Nevertheless, LMT is more promising to achieve this goal than the compared techniques when extending the testing time.

We further analyzed the reason why LMT significantly outperforms the compared techniques. One major reason is that LMT has a larger chance to generate valid test inputs than the others, which can promote our elaborately generated test inputs with the purpose of improving last-mile functional coverage to take effect. As presented in Section II, the embedded rules inside the constraint checker of the simulator can modify invalid test inputs to be valid for enabling the simulation process, but also limit the ability of improving functional coverage by making the modified inputs cover a fixed and limited set of functionality points. Hence, for each technique, we measured the percentage of valid cases (those do not have to be modified to be valid) among the 30,000 test inputs. The results are shown in Table II. From this table, the percentage of valid cases generated by LMT is 65.05% and 86.30% for $\mathcal{MA}$ and $\mathcal{MB}$ respectively, while those by random test input generation, DL-based and GA-based techniques are 33.72%, 35.19%, 39.40% on $\mathcal{MA}$ and 51.94%, 52.16%, 53.39% on $\mathcal{MB}$. This is because all the three compared techniques do not include the mechanism to learn complex constraints like the GAN model in LMT. In particular, GA-based test input generation performs even slightly worse than random test input generation in terms of last-mile functional coverage. This is because the former generating new test inputs highly depends on existing ones, while the latter does not. Hence, under the scenario with such complex constraints, the random test input generation technique can possibly explore a wider search space by generating more diverse test inputs.

Note: During the period of writing this paper, LMT has been deployed on six CDIs in Huawei as the standard testing technique (replacing the previous practice, i.e., random test input generation). According to the feedback from the CDI testing team, on the six CDIs, LMT saves 28.67% $\sim$ 83.79% testing time than the previous practice to achieve the same (high) functional coverage. This further confirms the effectiveness and generality of LMT.

C. RQ2: Contributions of LMT Components

1) Setup: LMT has two main components: 1) the RF model for identifying relevant variables to a targeted functionality group, which can significantly reduce the search space, and 2) the GAN model for learning to generate valid test inputs (satisfying complex constraints) as much as possible. To study the contribution of each of them to the overall effectiveness of LMT, we constructed two variants of LMT in this experiment: **LMT$_{noRF}$** (which removes the RF model from LMT and then randomly selects K relevant variables for a targeted functionality group) and **LMT$_{noGAN}$** (which removes the GAN model from LMT and randomly sets the remaining variables except the identified relevant variables by the RF model).

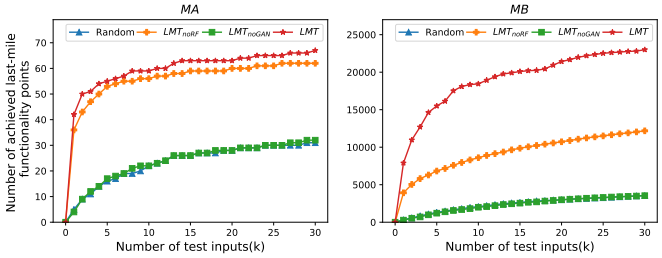


Fig. 6: Comparison between LMT, LMT$_{noRF}$ and LMT$_{noGAN}$

2) Results and Analysis: Figure 6 shows the comparison results between LMT and its two variants in terms of last-mile functional coverage, and Table II shows their comparison results in terms of the percentage of valid cases among the generated test inputs. We found that LMT always achieves much higher last-mile functional coverage than both LMT$_{noRF}$ and LMT$_{noGAN}$ on both CDIs regardless of the number of generated test inputs, demonstrating that **both components make large contributions to the effectiveness of LMT**. After running all the 30,000 test inputs, LMT achieves 97.10% and 88.77% last-mile functional coverage on $\mathcal{MA}$ and $\mathcal{MB}$ respectively, while LMT$_{noRF}$ achieves 89.86% and 47.03% and LMT$_{noGAN}$ achieves 46.38% and 13.66%.

Besides, on $\mathcal{MA}$, LMT$_{noRF}$ spent 12.54 hours (including 0.32 hours on building the GAN model) on running all the 30,000 generated test inputs for achieving 89.86% last-mile functional coverage, while LMT spent only 5.9 hours (including 0.35 hours on building RF and GAN models) on achieving the same coverage. Also, LMT$_{noGAN}$ spent 11.83 hours (including 0.03 hours on building the RF model) for achieving 46.38% last-mile functional coverage, while LMT spent only 0.67 hours (including 0.35 hours on building RF and GAN models) on achieving the same coverage. Overall, LMT saves 52.95% and 94.35% CDI testing time compared with LMT$_{noRF}$ and LMT$_{noGAN}$ on $\mathcal{MA}$, respectively. Similarly, on $\mathcal{MB}$, LMT saves 78.09% and 85.63% CDI testing time compared with LMT$_{noRF}$ and LMT$_{noGAN}$, respectively. The results further demonstrate the significant value of both components in LMT.

In particular, the GAN model contributes more than the RF model on both CDIs. The major reason also lies in that LMT$_{noRF}$ makes a larger chance to produce valid test inputs due to the existence of the GAN model, while LMT$_{noGAN}$ cannot. From Table II, the percentage of valid cases among the generated test inputs by LMT$_{noRF}$ is 63.58% while that by LMT$_{noGAN}$ is only 33.71% on $\mathcal{MA}$. Similarly, on $\mathcal{MB}$, the percentage of valid cases by LMT$_{noRF}$ is 84.45% while that by LMT$_{noGAN}$ is only 51.98%. The results indicate that the GAN model has indeed learned the constraints among input variables

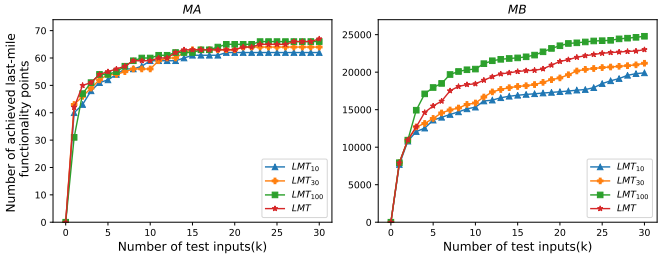


Fig. 7: Effectiveness of LMT under different K settings

and significantly improved the possibility of generating valid inputs. We also found that LMT$_{noGAN}$ performs very similarly with random test generation. This is as expected, since the number of identified relevant variables is much smaller than the number of the remaining variables and the remaining variables are randomly set in LMT$_{noGAN}$. That is, the variables that are set randomly dominate the testing capability of a generated test input by LMT$_{noGAN}$, which is very likely to violate the complex constraints like random test input generation.

D. RQ3: Influence of K

1) Setup: In LMT, the number of identified relevant variables (i.e., K) is the most important parameter, as it directly controls the size of search space for inputs and thus affects its effectiveness. Hence, it is important to study its influence on the effectiveness of LMT. In this experiment, besides the default setting in LMT (i.e., 50), we further studied three other K values, i.e., 10, 30, and 100. For ease of presentation, we call them **LMT**$_{10}$, **LMT**$_{30}$, and **LMT**$_{100}$, respectively.

2) Results and Analysis: Figure 7 shows the comparison results under different settings of K. First of all, regardless of the settings of K, our proposed technique significantly outperforms all the three compared techniques (from both Figure 5 and this figure), demonstrating the stable effectiveness of LMT. Then, we found that K is indeed able to affect the effectiveness of LMT, especially on $\mathcal{MB}$. From the Figure 7, LMT$_{100}$ and LMT perform better than LMT$_{10}$ and LMT$_{30}$ on $\mathcal{MA}$, and LMT performs slightly better than LMT$_{100}$. As for $\mathcal{MB}$, LMT$_{100}$ performs the best and LMT takes the second place. That is, different CDIs have different optimal settings of K, which is as expected as different characteristics in different CDIs could lead to different numbers of relevant variables. For our studied settings of K, the larger K values (i.e., 50 and 100) perform better than the smaller ones (i.e., 10 and 30), but meanwhile the larger K values could lead to the larger search space and thus may incur more time cost for exploring it. Hence, we recommend $K = 50$ as the default setting in LMT by balancing both effectiveness and efficiency. In particular, our experiments have demonstrated that LMT with the default setting indeed performs stably and well. In practice, testers can also adjust the setting of K by themselves according to the concerned metrics on the CDI under test.

V. Threats to Validity

The threat to *internal* validity mainly lies in the implementation. To reduce this kind of threat, two authors and industrial partners have carefully checked all the code. Also, they wrote unit tests to test the implementation of LMT. In addition, the intention of the RF model in LMT is to learn the relevance between input variables and the targeted functionality group, and thus its performance can be affected when there is no test input covering the targeted group in training data. However, this case is indeed rare in practice according to the feedback from our industrial partners, and it neither exists in the two real-world CDIs used in our study.

The threat to *external* validity mainly lies in the subjects used in our study. We evaluated the effectiveness of LMT on two CDIs from Huawei, which may not represent the CDIs in other companies. In our study, our used CDIs are industrial-grade and real-world, and they have very different characteristics. In particular, LMT has been deployed in Huawei for practical use on six CDIs. These can help reduce this kind of threat. In the future, we will evaluate LMT on more diverse CDIs to further reduce it.

The threat to *construct* validity mainly lies in the randomness in our study. To reduce this kind of threat, we repeated the experiments five times and calculated the average results. The times of repeating experiments may be not enough due to the limited computing resources. In the future, we will repeat our experiments more times.

VI. Related Work

CDI functional testing. Our work belongs to test input generation in CDI functional testing, and thus we mainly present the related work in this area. In the literature, the test input generation techniques in CDI functional testing can be divided into two main categories, i.e., search-based techniques [14]–[18] and DL-based techniques [19]–[22].

Regarding the former category, GA is the most widely-studied one for test input generation in CDI functional testing (and traditional software testing) [14]–[18]. For example, Subedha et al. [14] proposed to improve the overall coverage based on GA, which used statement and branch coverage of the CDI to design the fitness function. Yang et al. [18] proposed a GA-based CDI testing platform that adopts functional coverage as the fitness function to improve testing efficiency. In recent years, a number of DL-based test input generation techniques have been proposed in CDI functional testing, which mainly learn the relationship between test inputs and functional coverage. For example, Gogri et al. [22] used the learned relationship to filter out the test inputs without gains in functional coverage, in order to speed up the testing process. Gal et al. [21] used the learned relationship and derivative free optimization to increase the hitting frequency of the functionality points that have been covered *infrequently*.

Our work belongs to the second category. Different from them, LMT adopts RF to model the relevance between variables and each last-mile functionality group to identify relevant variables, and adopts GAN to learn to generate test inputs satisfying complex constraints among variables with a larger possibility without relying on domain knowledge and internal information of the CDI.

Traditional software test input generation. Many test input generation techniques have been proposed for traditional software, e.g., symbolic execution [28]–[31], [62]and fuzzing [32]–[34], [63]–[68]. Also, there are some testing techniques for highly-configurable software or software product lines, which also involve relatively large input space [69]–[73]. For example, Qu et al. [69] proposed a technique based on combinatorial interaction testing to generate configurations in configurable software regression testing. Cohen et al. [70] combined combinatorial interaction testing with SAT to generate configurations for highly-configurable software in the presence of constraints. However, as presented in Section I, they are not applicable to CDI functional testing due to 1) the scalability issue, 2) relying on domain knowledge or program internal information, or 3) lack of tools supporting the analysis of programs in hardware description languages.

Moreover, the techniques without considering complex constraints cannot perform well in CDI functional testing according to our evaluation. For example, adaptive random testing can also perform poorly in CDI functional testing since it has a hypothesis where test inputs are evenly distributed among the input space [74]. However, this hypothesis may not hold due to the complicated constraints in CDI functional testing. Besides, there are some invariant mining methods in traditional software [75]–[77], but similarly, they are hard to mine invariants from the entire black-box CDI due to the above reasons. These invariants are also hard to guide the construction of valid inputs in CDI functional testing, while our GAN model can provide a generator to facilitate the test input generation task.

VII. Conclusion

We propose the first test input generation technique, called LMT, targeting the challenge of improving *last-mile functional coverage* in CDI functional testing. LMT builds the RF and GAN models for identifying relevant variables to each last-mile functionality group, which can help largely reduce the input space, and learning to generate test inputs satisfying complex constraints among variables, respectively. Our study results on two industrial CDIs show that LMT significantly outperforms the state-of-the-art CDI test input generation techniques, demonstrating its effectiveness and efficiency.

Acknowledgment

We thank the anonymous reviewers for their constructive suggestions to help improve the quality of this paper. This work was supported by Huawei Project Funding.

References

[1] W. Ertel, *Introduction to artificial intelligence*. Springer, 2018.
[2] A. Gupta and R. K. Jha, "A survey of 5g network: Architecture and emerging technologies," *IEEE access*, vol. 3, pp. 1206–1232, 2015.
[3] "News," Accessed: 2022, https://en.wikipedia.org/wiki/Pentium_FDIV_bug.
[4] "News," Accessed: 2022, https://www.wired.com/story/apples-m1-chip-has-fascinating-flaw/.
[5] "News," Accessed: 2022, https://threatpost.com/qualcomm-chip-bug-android-eavesdropping/165934/.
[6] J. Bergeron, *Writing testbenches: functional verification of HDL models*. Springer Science & Business Media, 2012.
[7] B. Wile, J. Goss, and W. Roesner, *Comprehensive functional verification: The complete industry cycle*. Morgan Kaufmann, 2005.
[8] A. Piziali, *Functional verification coverage measurement and analysis*. Springer Science & Business Media, 2007.
[9] A. Evans, A. Silburt, G. Vrckovnik, T. Brown, M. Dufresne, G. Hall, T. Ho, and Y. Liu, "Functional verification of large asics," in *Proceedings of the 35th Annual Design Automation Conference*, 1998, pp. 650–655.
[10] P. Mishra and N. Dutt, "Functional coverage driven test generation for validation of pipelined processors," in *Design, Automation and Test in Europe*. IEEE, 2005, pp. 678–683.
[11] H. D. Foster, "Trends in functional verification: A 2014 industry study," in *Proceedings of the 52nd Annual Design Automation Conference*, 2015, pp. 1–6.
[12] J. Bhadra, M. S. Abadir, L.-C. Wang, and S. Ray, "A survey of hybrid techniques for functional verification," *IEEE Design & Test of Computers*, vol. 24, no. 02, pp. 112–122, 2007.
[13] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using bayesian networks," in *Proceedings of the 40th annual Design Automation Conference*. ACM, 2003, pp. 286–291.
[14] V. Subedha and S. Sridhar, "An efficient coverage driven functional verification system based on genetic algorithm," *European Journal of Scientific Research*, vol. 81, no. 4, pp. 533–542, 2012.
[15] A. Martínez-Cruz, R. Barrón-Fernández, H. Molina-Lozano, M.-A. Ramírez-Salinas, L.-A. Villa-Vargas, P. Cortés-Antonio, and K.-T. T. Cheng, "An automatic functional coverage for digital systems through a binary particle swarm optimization algorithm with a reinitialization mechanism," *Journal of Electronic Testing*, vol. 33, no. 4, pp. 431–447, 2017.
[16] H. Bhargav, V. Vs, B. Kumar, and V. Singh, "Enhancing testbench quality via genetic algorithm," in *2021 IEEE International Midwest Symposium on Circuits and Systems*. IEEE, 2021, pp. 652–656.
[17] J. Wang, Z. Liu, S. Wang, Y. Liu, Y. Li, and H. Yang, "Coverage-directed stimulus generation using a genetic algorithm," in *2013 International SoC Design Conference*. IEEE, 2013, pp. 298–301.
[18] R. Yang, L. Wu, J. Guo, and B. Liu, "The research and implement of an advanced function coverage based verification environment," in *2007 7th International Conference on ASIC*. IEEE, 2007, pp. 1253–1256.
[19] M. Fajcik, P. Smrz, and M. Zachariasova, "Automation of processor verification using recurrent neural networks," in *2017 18th International Workshop on Microprocessor and SOC Test and Verification*. IEEE, 2017, pp. 15–20.
[20] R. Gal, E. Haber, B. Irwin, M. Mouallem, B. Saleh, and A. Ziv, "Using deep neural networks and derivative free optimization to accelerate coverage closure," in *2021 ACM/IEEE 3rd Workshop on Machine Learning for CAD*. IEEE, 2021, pp. 1–6.
[21] R. Gal, E. Haber, and A. Ziv, "Using dnns and smart sampling for coverage closure acceleration," in *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*. ACM, 2020, pp. 15–20.
[22] S. Gogri, J. Hu, A. Tyagi, M. Quinn, S. Ramachandran, F. Batool, and A. Jagadeesh, "Machine learning-guided stimulus generation for functional verification," in *Proceedings of the Design and Verification Conference*, 2020, pp. 2–5.
[23] J. Yuan, C. Pixley, A. Aziz, and K. Albin, "A framework for constrained functional verification," in *2003 International Conference on Computer-Aided Design*. IEEE, 2003, pp. 142–145.
[24] M. Grindal, J. Offutt, and S. F. Andler, "Combination testing strategies: a survey," *Software Testing, Verification and Reliability*, vol. 15, no. 3, pp. 167–199, 2005.
[25] Y. Yao, J. Liu, J. Zhu, and X. Gu, "Combinational metamorphic testing in integer vulnerabilities detection," *Journal of Computational Methods in Sciences and Engineering*, vol. 20, no. 4, pp. 1053–1061, 2020.
[26] C. Wang, W. Ge, X. Li, and Z. Feng, "Dct: Differential combination testing of deep learning systems," in *Artificial Neural Networks and Machine Learning–ICANN 2019: Image Processing: 28th International Conference on Artificial Neural Networks, Munich, Germany, September 17–19, 2019, Proceedings, Part III 28*. Springer, 2019, pp. 697–710.
[27] L. S. Ghandehari, Y. Lei, R. Kacker, R. Kuhn, T. Xie, and D. Kung, "A combinatorial testing-based approach to fault localization," *IEEE Transactions on Software Engineering*, vol. 46, no. 6, pp. 616–645, 2018.
[28] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, 2013.
[29] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[30] C. Hsin-Yu, H. Chin-Yu, and C. Fang, "Applying slicing-based testability transformation to improve test data generation with symbolic execution," *International Journal of Performability Engineering*, vol. 17, no. 7, p. 589, 2021.

[31] W. He, J. Shi, T. Su, Z. Lu, L. Hao, and Y. Huang, "Automated test generation for IEC 61131-3 ST programs via dynamic symbolic execution," *Science of Computer Programming*, vol. 206, p. 102608, 2021.

[32] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proceedings of the Network and Distributed System Security Symposium*, vol. 8. The Internet Society, 2008, pp. 151–166.

[33] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 474–484.

[34] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *8th USENIX Symposium on Operating Systems Design and Implementation*, vol. 8, 2008, pp. 209–224.

[35] H. Zou and T. Hastie, "Regularization and variable selection via the elastic net," *Journal of the royal statistical society: series B (statistical methodology)*, vol. 67, no. 2, pp. 301–320, 2005.

[36] G. Chandrashekar and F. Sahin, "A survey on feature selection methods," *Computers & Electrical Engineering*, vol. 40, no. 1, pp. 16–28, 2014.

[37] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 58, no. 1, pp. 267–288, 1996.

[38] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," *Advances in neural information processing systems*, vol. 27, 2014.

[39] A. B. Mehta, "Asic/soc functional design verification," in *A Comprehensive Guide To Technologies and Methodologies*. Springer, 2018.

[40] M. Teplitsky, A. Metodi, and R. Azaria, "Coverage driven distribution of constrained random stimuli," in *Proceedings of the design and verification conference and exhibition US (DVCon)*, 2015.

[41] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[42] R. Genuer, J.-M. Poggi, and C. Tuleau-Malot, "Variable selection using random forests," *Pattern recognition letters*, vol. 31, no. 14, pp. 2225–2236, 2010.

[43] J. L. Speiser, M. E. Miller, J. Tooze, and E. Ip, "A comparison of random forest variable selection methods for classification prediction modeling," *Expert systems with applications*, vol. 134, pp. 93–101, 2019.

[44] J. Chen, N. Xu, P. Chen, and H. Zhang, "Efficient compiler autotuning via bayesian optimization," in *2021 IEEE/ACM 43rd International Conference on Software Engineering*. IEEE, 2021, pp. 1198–1209.

[45] B. H. Menze, B. M. Kelm, R. Masuch, U. Himmelreich, P. Bachert, W. Petrich, and F. A. Hamprecht, "A comparison of random forest and its gini importance with standard chemometric methods for the feature selection and classification of spectral data," *BMC Bioinform.*, vol. 10, 2009.

[46] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2016, pp. 785–794.

[47] J. G. Dy and C. E. Brodley, "Feature selection for unsupervised learning," *Journal of machine learning research*, vol. 5, no. Aug, pp. 845–889, 2004.

[48] A. Creswell, T. White, V. Dumoulin, K. Arulkumaran, B. Sengupta, and A. A. Bharath, "Generative adversarial networks: An overview," *Signal Process. Mag.*, vol. 35, no. 1, pp. 53–65, 2018.

[49] I. Goodfellow, "Nips 2016 tutorial: Generative adversarial networks," *arXiv preprint arXiv:1701.00160*, 2016.

[50] D. G. Altman and J. M. Bland, "Statistics notes: the normal distribution," *Bmj*, vol. 310, no. 6975, p. 298, 1995.

[51] "Random forest classifier in scikit-learn," Accessed: 2022, https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html.

[52] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. C. Courville, "Improved training of wasserstein gans," *Advances in neural information processing systems*, vol. 30, 2017.

[53] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *arXiv preprint arXiv:1511.06434*, 2015.

[54] M. Mirza and S. Osindero, "Conditional generative adversarial nets," *arXiv preprint arXiv:1411.1784*, 2014.

[55] "Homepage," Accessed: 2022, https://github.com/Jacob-yen/LMT.

[56] B. Xu, X. Xie, L. Shi, and C. Nie, "Application of genetic algorithms in software testing," in *Advances in Machine Learning Applications in Software Engineering*. IGI Global, 2007, pp. 287–317.

[57] P. R. Srivastava and T.-h. Kim, "Application of genetic algorithm in software testing," *International Journal of software Engineering and its Applications*, vol. 3, no. 4, pp. 87–96, 2009.

[58] M. Harman, Y. Jia, and Y. Zhang, "Achievements, open problems and challenges for search based software testing," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation*. IEEE, 2015, pp. 1–12.

[59] R. Lefticaru and F. Ipate, "Automatic state-based test generation using genetic algorithms," in *Ninth international symposium on symbolic and numeric algorithms for scientific computing*. IEEE, 2007, pp. 188–195.

[60] A. F. Gad, "Pygad: An intuitive genetic algorithm python library," 2021.

[61] J. Czerwonka, "Pairwise testing in real world," in *24th Pacific Northwest Software Quality Conference*, vol. 200. Citeseer, 2006.

[62] J. Chen, W. Hu, L. Zhang, D. Hao, S. Khurshid, and L. Zhang, "Learning to accelerate symbolic execution via code transformation," in *32nd European Conference on Object-Oriented Programming*, ser. LIPIcs, vol. 109. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, pp. 6:1–6:27.

[63] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy*. IEEE, 2018, pp. 679–696.

[64] L. Zhao, Y. Duan, H. Yin, and J. Xuan, "Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing." in *26th Annual Network and Distributed System Security Symposium*, 2019.

[65] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *21st USENIX Security Symposium*, 2012, pp. 445–458.

[66] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang, "Deep learning library testing via effective model generation," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2020, pp. 788–799.

[67] Y. Zhao, Z. Wang, J. Chen, M. Liu, M. Wu, Y. Zhang, and L. Zhang, "History-driven test program synthesis for JVM testing," in *44th IEEE/ACM 44th International Conference on Software Engineering*. ACM, 2022, pp. 1133–1144.

[68] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," *ACM Computing Surveys*, vol. 53, no. 1, pp. 4:1–4:36, 2021.

[69] X. Qu, M. B. Cohen, and G. Rothermel, "Configuration-aware regression testing: an empirical study of sampling and prioritization," in *Proceedings of the 2008 international symposium on Software testing and analysis*, 2008, pp. 75–86.

[70] M. B. Cohen, M. B. Dwyer, and J. Shi, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach," *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 633–650, 2008.

[71] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. Le Traon, "Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 650–670, 2014.

[72] J. Chen and C. Suo, "Boosting compiler testing via compiler optimization exploration," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 4, pp. 72:1–72:33, 2022.

[73] J. Chen, C. Suo, J. Jiang, P. Chen, and X. Li, "Compiler test-program generation via memoized configuration search," in *2023 IEEE/ACM 45th International Conference on Software Engineering*. IEEE, 2023, to appear.

[74] T. Y. Chen, H. Leung, and I. K. Mak, "Adaptive random testing," in *ASIAN*, ser. Lecture Notes in Computer Science, vol. 3321. Springer, 2004, pp. 320–329.

[75] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1–3, pp. 35–45, Dec. 2007.

[76] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest, "Dig: a dynamic invariant generator for polynomial and array invariants," *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 4, pp. 1–30, 2014.

[77] L. Zhang, G. Yang, N. Rungta, S. Person, and S. Khurshid, "Feedback-driven dynamic invariant discovery," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 362–372.