Twin-Finder: Integrated Reasoning Engine for Pointer-Related Code Clone Detection

Hongfa Xue
The George Washington University
Washington, DC, USA
Email:hongfaxue@gwu.edu

Yongsheng Mei
The George Washington University
Washington, DC, USA
Email: ysmei@gwu.edu

Kailash Gogineni
The George Washington University
Washington, DC, USA
Email: kailashg26@gwu.edu

Guru Venkataramani
The George Washington University
Washington, DC, USA
Email: guruv@gwu.edu

Tian Lan
The George Washington University
Washington, DC, USA
Email: tlan@gwu.edu

***Abstract*—Detecting code clones is crucial in various software engineering tasks. In particular, code clone detection can have significant uses in the context of analyzing and fixing bugs in large scale applications. However, prior works, such as machine learning-based clone detection, may cause a considerable amount of false positives. In this paper, we propose *Twin-Finder*, a novel, closed-loop approach for pointer-related code clone detection that integrates machine learning and symbolic execution techniques to achieve precision. *Twin-Finder* introduces a clone verification mechanism to formally verify if two clone samples are indeed clones and a feedback loop to automatically generated formal rules to tune machine learning algorithm and further reduce the false positives. Our experimental results show that *Twin-Finder* can swiftly identify up 9× more code clones comparing to a tree-based clone detector, Deckard and remove an average 91.69% false positives.**

***Index Terms*—Memory Safety, Formal methods, Code Clones**

I. Introduction

With rapid rise in software sizes and complexity, analyzing and fixing bugs in large scale applications is becoming increasingly critical. Similar code fragments are common in large code bases [16], [6]. Detecting such code fragments, usually referred as *code clones*, is crucial in various software engineering tasks, such as vulnerability discovery, refactoring and plagiarism detection. Prior works that use subsequence matching [5] have shown good performance in detecting *text-based similar code clones*. They have limited scalability since the pairwise string or tree comparison is expensive in large code bases. Code clone detection using machine learning approaches, such as clustering algorithms, improves the previous string-matching based clone detections. However, this may still cause a considerable amount of false positives.

In this paper, we introduce a novel clone detection approach, **Twin-Finder**, that is designed for better clone detection. Our approach uses domain-specific knowledge for code clone analysis, which can be used to detect code clone samples spanning non-contiguous and intertwined code base in software applications. We design and demonstrate our framework for pointer-related code clone detection, as pointers and pointer-related operations widely exist in real-world applications [7] and often cause security bugs, detecting such pointer-specific code clones are of great significance.

To verify the robustness of detection, we design a clone verification mechanism using symbolic execution (SE) that formally verifies if the two clone samples are indeed true code clones. Existing works have reported that the false positives from code clone detection are inevitable and human efforts are still needed for further verification and tuning detection algorithms [16]. To automate this verification process, we introduce a feedback loop using formal analysis. We compare the Abstract Syntax Trees (AST) representing two code clone samples if we observe they have different memory safety conditions. We add numerical weight to the feature vectors corresponding to the two code clone samples, based on the outputs from the tree comparison. Finally, we exponentially recalculate the distances among feature vectors to reduce the false positives admitted from code clone detection.

The contributions of this paper are summarized as follows:

- We propose **Twin-Finder**, a pointer-related code clone detection framework. **Twin-Finder** can automatically identify related codes from large code bases and perform code clone detection.
- **Twin-Finder** leverages program slicing to remove irrelevant codes and isolate analysis targets to find non-contiguous and intertwined clones. Our evaluation demonstrates that **Twin-Finder**can detect up to 9× more clones comparing to Deckard.
- **Twin-Finder** deploys formal analysis to perform a closed-loop operation. In particular, **Twin-Finder** introduces a clone verification mechanism to formally verify if two clone samples are indeed clones and a feedback loop to tune code clone detection algorithm and further reduce an average 91.69% false positives.

II. Problem Statement and Motivation

A. Challenges

Assuming we want to detect code clones for pointer-related code clones, existing code clone detection approaches are inefficient for this purpose, due to the considerable amount of pointer-irrelevant codes coupled with the target pointers. Even most advanced deep learning approaches currently *fail* to extract clone samples where pointer-related codes are intertwined with other codes. Another issue from current clone detection approaches is that they cannot guarantee zero false positives. To eliminate false positives, it always requires human efforts for further verification. If we can eliminate the false positives

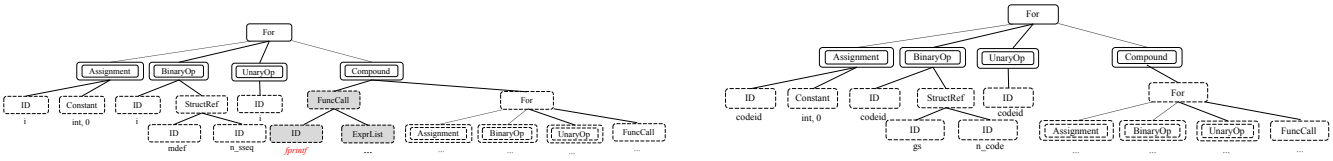
```
1  void dict2pid_dump (...){
2  ...
3  for (i = 0; i < mdef->n_sseq; i++) {
4    fprintf (fp, "%5d ", i);
5    for (j = 0; j < mdef_n_emit_state(mdef); j
       ++)
6      fprintf (fp, "%5d", mdef->sseq[i][j]);
7  ..
8  }
9  ..
10 }
```

Code fragment of function *sphinx3::dict2pid_dump* as pointer $\{mdef->sseq\}$ are intertwined inside of the function

```
1  int32 gc_compute_closest_cw (...){
2  ...
3  for(codeid=0; codeid< gs->n_code ;codeid+=2){
4    for(cid=0;cid<gs->n_featlen ; cid++)
5      fprintf (fp, "%5d", gs->codeword[codeid][cid])
       ;
6  }
7  ...
8  }
9  ...
10 }
```

Code fragment of function *sphinx3::gc_compute_closest_cw* as pointer $\{gs->codeword\}$ are intertwined inside of the function

Fig. 1. A true positive example



(a) AST of function *sphinx3::dict2pid_dump*

(b) AST of function *sphinx3::gc_compute_closest_cw*

Fig. 2. ASTs generated from the true positive example in Figure 1, where the shady nodes represent the different nodes between two trees

```
1  int32 mgau_eval (..., int32 *active)
2  {
3  ...
4  for (j = 0; active[j] >= 0; j++) {
5    c = active[j];
6    ...
7    }
8  ...
9  }
```

```
1  void lextree_hmm_histbin (lextree_t *lextree
       ,...)
2  {
3  ...
4  for (i = 0; i < lextree->n_active; i++) {
5    ln = list[i];
6    ..
7    }
8  ...
9  }
```

Code clone samples of function *sphinx3::mgau_eval* and *sphinx3::lextree_hmm_histbin* as pointer $\{active\}$ and $\{list\}$ are intertwined inside of the functions

Fig. 3. A false positive example

as many as possible, We still can enable a better analysis with more clone samples.

B. Motivating Example

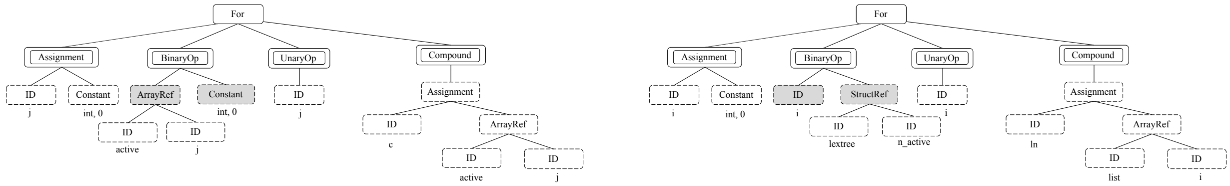
We use real-world false positive and true positive samples in sphinx3 from SPEC2006 benchmark reported from a tree-based code clone detector DECKARD [5] as motiving examples. First, we give the formal definition of false positive which is defined in Definition 1.

Definition 1. ***False Positives.*** *In this paper, we define false positives occur if a code clone pair is identified as code clones by code clone detection, but two clone samples share different bound safety constraints in terms of pointer analysis.*

Conventional clone detections, such as combining tree-based approach with machine learning techniques, introduce a code similarity measurement S and transfer the code into intermediate representations (e.g. Abstract Syntax Trees (ASTs)) to detect more code clones. This can help to detect clones that are not identical but still sharing a similar code structure. Consider the true positive example in Figure 1, in tree-based clone detection, two source files are first parsed and converted into Abstract Syntax Trees (ASTs), where all identifier names and literal values are replaced by AST nodes. For example, the initialization and exit conditions in *for* loops are replaced as **Assignment, BinaryOp, UnaryOp** and so on. Then a tree pattern is generated from post-order tree traversal. After, a pairwise tree pattern comparison can be used to detect such clones. In Figure 2 we plot the ASTs for these two clone samples correspondingly. Both ASTs share a common tree pattern with only three different nodes appeared in the first code sample. It is clear to see that the first code sample has an extra function call *fprintf* compared to the second code sample. If we relax the code similarity threshold, these two code samples are identified as code clones.

To proceed with a dependency analysis process, variables$\{i, j, mdef->n_sseq, mdef_n_emit_state(mdef)\}$ are identified as pointer-related variables (that can potentially affect the value of pointers) for target pointer $\{mdef->sseq\}$ in the first example (second code example is applied with the same procedure). However, *fprintf* cannot affect any values of those variables. Thus, the bound safety conditions can be simply derived as these two equations.

$$\{i < length(mdef->sseq)\} \wedge \{j < length(*mdef->sseq)\} \quad (1)$$



(a) AST of function *sphinx3::mgau_eval* (b) AST of function *sphinx3::lextree_hmm_histbin*

Fig. 4. ASTs generated from false positive example in Figure 3, where the shady nodes represent the different nodes between two trees

$$\{codeid < length(gs- > codeword)\} \wedge \{cid < length(*gs- > codeword)\} \quad (2)$$

respectively. As we can see, they are identical because the conditions differ only in variable names. Thus, they are true positives as they share the same pointer safety conditions.

Even though a relaxed code similarity is able to detect such clones, it can also introduce a considerable amount of false positives. Figure 3 illustrates one false positive example detected in sphinx3 from SPEC2006 benchmark. Two *for* loops are identified as code clones under a certain code similarity threshold. Figure 4 shows the ASTs generated from those two code samples respectively. As we can see, they indeed share a common tree pattern but with 2 different nodes in shady color. Even though they are not identical, they still can be identified as similar looking code clones if we relax the code similarity threshold. Assuming the target pointers for analysis are $active$ and $list$, we first obtain pointer related variables through dependency analysis. It is easy to see that a solely variable j is related to pointer $active$ but two variables $\{i, lextree- > n\_active\}$ are related to $list$. Thus, the bound safety conditions are deemed different. As mentioned in Definition 1, these two code clones will be defined as false positives since they do not share the same safety conditions.

III. System Design

In this section, we present details of our Twin-finder and show how our system is designed. Two main components of **Twin-Finder** are shown in Figure 5, namely Domain Specific Slicing and Closed-loop Code Clone Detection.

A. Domain Specific Slicing

For pointer analysis, only some certain types of variables are related to the target pointer for further consideration, which can affect the base, offset or bound information of this pointer (such as array index, pointer increment and other similar types of variables). Here, we name such variables as *pointer-related variables*. Analyzing only pointers in the programs requires unrelated codes to be discarded automatically. However, this selection of relevant codes requires the knowledge of control flow and dependency of data among pointer-related variables to be taken into account. To address this problem, **Twin-Finder** first performs dependency analysis of the code and deploy program slicing to isolate pointer related code in three steps:

1) **Pointer Selection.** Given a source code of a program, we utilize the static code analysis to select all the pointers and collect related information from the code, including variable name, pointer declaration type (e.g. global variables, local variables or structures) and the location in the code (defined and used in which function). The types of selected pointers consist of the pointers/arrays defined as local/global variables, the elements of structures and function parameters. We generate a pointer list for each program through such pointer selection process, denoted as $PtrList = \{p_1, p_2, \ldots, p_m\}$, where p_i represents a target pointer for further analysis (for $i = 1, \ldots, m$).
2) **Dependency Analysis and Lightweight Tainting** A directed dependency graph $\mathcal{DG} = (\mathcal{N}, \mathcal{E})$ is created for each pointer p_i within the function where it is originally declared. The nodes of the graph $\mathcal{N}$ represent the identifiers in the function and edges $\mathcal{E}$ represent the dependency between nodes, which reflects array indexing, assignments between identifiers and parameters of functions. As soon as the dependency graph is constructed, we start with the target pointer p_i and traverse the dependency graph to discover all pointer-related variables in both top-down and bottom-up directions. This tainting propagation process stops at function boundaries. In the end, we generate the pointer-related variable list $p_i = \{v_1, v_2, \ldots, v_n\}$, where v_i represents a pointer-related variable for pointer p_i.
3) **Isolating Code through Slicing.** We use both forward and backward program slicing to isolate code into pointer-isolated code. Given a pointer-related variable list $V = \{v_1, v_2, \ldots, v_n\}$ for a target pointer p_i, we first make use of backward slicing: we construct a backward slice on each variable $v_i \in V$ at the end of the function and slice backwards to only add the statements into slice iff there is data dependency as v_i is on left-hand side of assignments or parameter of functions, which can potentially affect the value of v_i, in the slice. Whenever v_i is in a loop (e.g. $while/for$ loop) or $if-else/switch$ branches, forward slicing is then used to add those control dependency statements to the slice.

B. Code Clone Detection

Twin-Finder leverages a tree-based code clone detection approach, which is originally proposed by Jiang et al. [5]. We adopt the notions of code similarity, feature vectors and other related definitions from previous work [5]. We deploy such method on the top of our domain specific slicing module to only detect code clones among pointer isolated codes.

1) Definitions: We first formally give the several definitions used in our code clone detection module.

Definition 2. ***Code Similarity.***

Given two Abstract Syntax Trees (AST) T_1 and T_2, which are representing two code fragments, the code similarity S between them is defined as:

$$S(T_1, T_2) = \frac{2S}{2S + L + R} \quad (3)$$

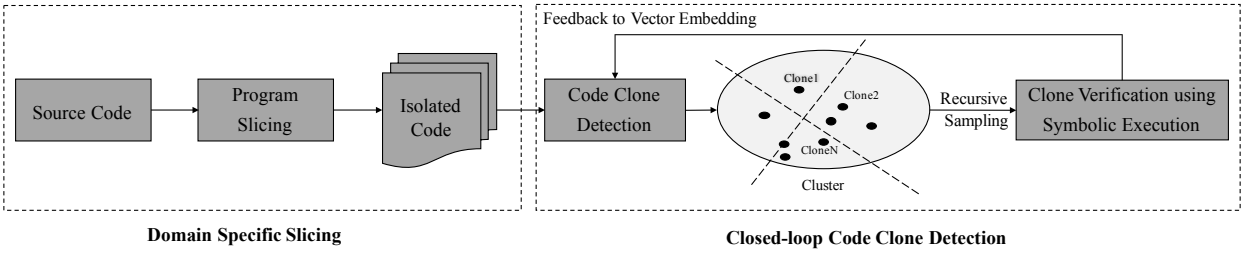


Fig. 5. Approach Overview

S is the number of shared nodes in T_1 and T_2; $\{L : [t_1, t_2, .., t_n], R : [t_1, t_2, .., t_m]\}$ are the different nodes between two trees, where t_i represents a single AST node.

Definition 3. ***Feature Vectors.*** *A feature vector $V = (v_1, v_2, ..., v_n)$ in the Euclidean space is generated from a sub-AST, corresponding to a code fragment, where each v_i represents a specific type of AST nodes and is calculated by counting the occurrences of corresponding AST node types in the sub-AST.*

Given an AST tree T , we perform a post-order traversal of T to generate vectors for its subtrees. Vectors for a subtree are summed up from its constituent subtrees.

2) Clone Detection: Given a group of feature vectors, we utilize Locality Sensitive Hashing (LSH) [4] and near-neighbor querying algorithm based on the euclidean distance between two vectors to cluster a vector group Suppose two feature vectors V_i and V_j representing two code fragments C_i and C_j respectively. The code size (the total number of AST nodes) are denoted as $S(C_i)$ and $S(C_j)$. The euclidean distance between V_i and V_j are denoted as $E([V_i; V_j])$ Then, given a feature vector group V, the threshold can be simplified as $\sqrt{2(1-S) \times min_{v \in V} \in S(v)}$, where we use vector sizes to approximate tree sizes. The S is the code similarity metric defined from Equation 2(we omit the details of how this threshold is derived. We refer readers to [5]). Thus, code fragments C_i and C_j will be clustered together as code clones under a given code similarity S if $E([V_i; V_j]) \leq T$.

C. Clone Verification

To *formally check* if the code clones detected by **Twin-Finder** are indeed code clones in terms of pointer memory safety, we propose a clone verification mechanism and utilize symbolic execution as our verification tool.

1) Recursive Sampling: To improve the coverage of code clone samples in the clusters, we propose a recursive sampling procedure to select clone samples for clone verification.

First, we randomly divide one cluster into several smaller clusters. Then we pick random code clone samples from each smaller cluster center and cluster boundary. After, we employ symbolic execution in selected samples for further clone verification. Note that the code clone samples are pointer isolated code generated from program slicing. Since symbolic execution requires the code completeness, we map the code clone samples to the original source code locations to perform symbolic execution.

2) Clone Verification: Clustering algorithm cannot offer any guarantees in terms of ensuring safe pointer access from all detected code clones. It is possible that two code fragments are clustered together, but have different bound safety conditions, especially if we use a smaller code similarity. To further improve the clone detection accuracy of Twin-finder, we design a clone verification method to check whether the code clone samples are true clones.

Let $X = \{p_1, p_2, ..., p_n\}$ be a finite set of pointer-related variables as symbolic variables, while symbolic executing a program all possible paths, each path maintains a set of *constraints* called the *path conditions* which must hold on the execution of that path. First, we define an atomic condition, $AC()$, over X is in the form of $f(p_1, p_2, ..., p_n)$, where f is a function that performs the integer operations on $O \in \{>, <, \geq, \leq, =\}$. Similarly, a condition over X can be a Boolean combination of path conditions over X.

Definition 4. ***Constraints.*** *An execution path can be represented as a sequence of basic blocks. Thus, path conditions can be computed as $AC(b_0) \wedge AC(b_1) ... \wedge AC(b_n)$ where each $AC(b_i)$ in $AC()$ represents a sequence of atomic condition in the basic block b_n. For the case of involving multiple execution paths, the final constraints will be the union of all path conditions.*

Example. Back to the example mentioned in Figure 1. The code fragment of function *sphinx3::dict2pid_dump* includes two *for* loops, representing two basic blocks (b_1, b_2). Thus, there are two paths in this code fragment. For the first *for* loop, we can derive an atomic condition $AC(b_1) = \{i < length(mdef- > sseq)\}$. Similarly, we can get the second condition of the second *for* loop as $AC(b_2) = \{j < length(*mdef- > sseq)\}$. Finally, the path conditions for this code can be computed as $AC(b_1) \wedge AC(b_2)$.

Give a clone pair sampled from the previous step, we perform symbolic execution from beginning to the end of clone samples in original source code based on the locations information (line numbers of code). The symbolic executor is used to explore all the possible paths existing in the code fragment. We collect all the possible constraints(defined in Definition 4) for each clone sample after symbolic execution is terminated. Then the verification process is straightforward. A constraint solver can be used to check the satisfiability and syntactic equivalence of logical formulas over one or more theories.

The steps of this verification process are summarized as follows:

- **Matching the Variables**: To verify if two sets of constraints are equal, we omit the difference of variable names. However, we need to match the variables between two constraints based on their dependency of target pointers.

For instance, two pointer dereference $a[i] =' A'$ and $b[j] =' B'$, the indexing variables are i and j respectively. During symbolic execution, they both will be replaced as symbolic variables, and we do not care much about the variables names. Thus, we can derive a precondition that i is equivalent to j for further analysis. This prior knowledge can be easily obtained through dependency analysis mentioned in Section 4.

- **Simplification**: Given a memory safety condition S, it can contain multiple linear inequalities. For simplicity, the first step is to find possibly simpler expression S', which is equivalent to S.
- **Checking the Equivalence**:To prove two sets of constraints $S_1 == S_2$,we only need to prove the negation of $S_1 == S_2$ is unsatisfiable.

Example. Assuming we have two sets of constraints, $S_1 = (x_1 \geq 4) \wedge (x_2 \geq 5)$ and $S_2 = (x_3 \geq 4) \wedge (x_4 \geq 5)$, where x_1 is equivalent to x_3 and x_2 is equivalent to x_4. We then can solve that $Not(S_1 == S_2)$ is unsatisfiable. Thus, $S_1 == S_2$.

D. Formal Feedback to Vector Embedding

Algorithm 1 Algorithm for Feedback to Vector Embedding

1: **Input:**: Code Clone Samples C_i, C_j
2: Corresponding AST sub-trees: S_i, S_j
3: Corresponding Feature Vectors: V_i, V_j
4: Current Code similarity threshold: S
5: Longest Common Subsequence **function:** LCS ()
6: **Output:**: Optimized Feature vectors: O_i, O_j
7: **Initialization:**
8: $O_i, O_j = V_i, V_j$
9: $D = LCS(S_i, S_j)$
10: **if** C_i and C_j share same constraints **then**
11: $S_i = RemoveSubtrees(S_i - D)$
12: $S_j = RemoveSubtrees(S_j - D)$
13: $O_{n\in\{i;j\}} = Vectornize(S_{n\in\{i;j\}})$;
14: **else**
15: T=[]
16: $Uncommon\_Subtrees = (S_i - D) + (S_j - D)$
17: $T.append(Uncommon\_Subtrees)$
18: **for** t in T **do**
19: **if** $EuclideanDistance(O_i, O_j) < \sqrt{2(1-S) \times min\{Size(V_i), Size(V_j)\}}$ **then**
20: $break$;
21: $t = d.index$
22: $O_{n\in\{i;j\}}[t] = O_{n\in\{i;j\}}[t] * \delta$; *where* $\delta > 1.0$

While using the formal method to verify if the two clone samples are true clones, we provide a feedback process to the vector embedding in code clone detection to reduce false positives. Since the code clone detection is based on the euclidean distance between data points over a code similarity threshold, the feedback is a mechanism to tune the feature vectors weights. Based on the constraints we obtained from symbolic execution, we are able to determine which type of variables or statements causing different constraints between two clone samples. We use such information to guide feedback to vector embedding in clone detection module. Now we describe a feedback mechanism to vector embedding in code clone detection if we observe false positives verified through the execution in Section III-C2.

To tune and adjust the weights in the feature vectors, we design an algorithm for our feedback. Algorithm 1 shows the steps of feedback in detail. Given a code similarity threshold S, It takes two clone samples (C_i, C_j), corresponding AST sub-trees (S_i, S_j) and feature vectors (V_i, V_j) representing two code clones as inputs (line 1-4 in Algorithm 1), and we utilize a helper function $LCS()$ to find the Longest Common Subsequence between two lists of sub-trees.

When the code clone samples are symbolically executed, we start by checking if the constraints, obtained from previous formal verification step, are equivalent. Then the feedback procedure after is conducted as two folds:

(1) If they indeed share the same constraints, we remove the uncommon subtrees (where can be treated as numerical weight as 0) as we now know they will not affect the output of constraints (line 10-13). This process is to make sure the remaining trees are identical so that they will be detected as code clone in the future.

(2) If they have different constraints, we obtain the uncommon subtrees from (S_i, S_j)(line 15-17) and add numerical weight, $\delta > 1.0$, one by one. We iterate the list and we trace back to the vector using the vector index to adjust the weight δ for that specific location correspondingly (line 18-22). We initialize the weight δ as a random number which is greater than 1.0 and re-calculate the euclidean distance between two feature vectors. We repeat this process until the distance is out of current code similarity threshold S (line 19-20). This is designed to guarantee that these two code samples will not be considered as code clone in the future. Finally, the feedback can run in a loop fashion to eliminate false positives. The termination condition for our feedback loop is that no more false positives can be further eliminated or observed.

Example: Here, we give an example to illustrate how our formal feedback works. We use the false positive example showing in Figure 4. Assuming the feature vectors are $< 7, 2, 2, 2, 0, 1, 1, 1, 1 >$ and $< 8, 1, 1, 2, 1, 1, 1, 1, 1 >$ respectively, where the ordered dimensions of vectors are occurrence counts of the relevant nodes: **ID, Constant, ArrayRef, Assignment, StrucRef, BinaryOp, UnaryOp, Compound,** and **For**. Based on the threshold defined in equation III-B, these two code fragments will be clustered as clones when $S = 0.75$. During the feedback loop, we first identify these 2 different nodes in each tree by finding the LCS. Assuming we initial the weight $\delta = 2$ and add it to the corresponding dimension in the feature vectors, we can obtain the updated feature vectors as $< 7, 1+1\times\delta, 1+1\times\delta, 1+1\times\delta, 0, 1, 1, 1, 1 >$ and $< 7+1\times\delta, 1, 1, 2, 1\times\delta, 1, 1, 1, 1 >$. We then re-calculate the euclidean distance of these two updated feature vectors, and they will be no longer satisfied within the threshold $\sqrt{2(1-S) \times min(S(C_i), S(C_j))}$. Thus, we can eliminate such false positives in the future.

We instrument a source code symbolic executor, KLEE [3] and SMT solver Z3 [20] for our clone verification module. We develop a python script for our formal feedback module.

IV. Evaluation

This section presents a detailed evaluation results of **Twin-Finder** against a tree-based code clone detection tool

DECKARD [5] in terms of code clone detection, and conduct several case studies for applications security analysis.

A. Experiment Setup

We selected 7 different benchmarks from real-world applications: bzip2, hmmer and sphinx3 from SPEC2006 benchmark suite [1]; man and gzip from Bugbench [9]; thttpd-2.23beat1 [2], a well-known lightweight sever and a lightweight browser links-2.14 [10].

B. Code Clones Detection

Benchmark	Program Size (LoC)	#Code clones without slicing and feedback	#Code clones Our approach	% Code clones
bzip2	5,904	432	1,084	150.92%
sphinx3	13,207	1,047	3,546	238.68%
hmmer	20,721	1,238	4,391	254.68%
thttpd	7,956	611	1,398	128.80%
gzip	5,225	36	365	913.89%
man	3,028	47	443	842.55%
links	178,441	3,007	9,809	226.21%

TABLE I
COMPARISON OF NUMBER OF CODE CLONES DETECTED BEFORE AND AFTER USING OUR APPROACH

We measure code clone quantity by the number of code clones that are detected before and after we use **Twin-Finder** for pointer analysis purpose. We conduct two experiments in terms of the following: code clones quantity, the flexibility of code similarity configuration and false positives analysis.

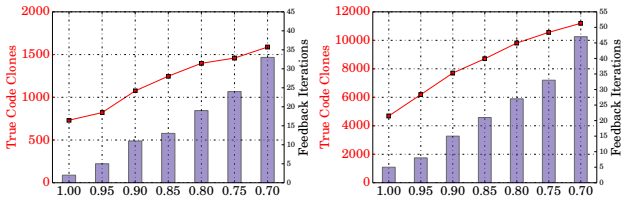
Benchmark	Pointer related Code LoC	Clone Detection w/ DECKARD		Clone Detection w/ Our Approach	
		# Cloned LoC	% Cloned LoC	# D.S LoC	% D.S LoC
bzip2	3,279	1,066	32.51%	2,038	62.15%
sphinx3	9,519	3,073	32.28%	7,224	75.89%
hmmer	11,635	3,163	27.19%	6,929	59.55%
thttpd	4,390	1,279	29.13%	2,267	51.64%
gzip	2,289	219	9.57%	919	40.15%
man	1,683	248	14.74%	826	49.08%
links	28,334	6,429	22.69%	18,334	64.71%

TABLE II
COMPARISON OF CODE CLONE COVERAGE BETWEEN DECKARD AND OUR APPROACH

Benchmark	# True Code Clones			# Feedback Iterations		
	S = 1.0	S = 0.90	S = 0.80	S = 1.0	S = 0.90	S = 0.80
bzip2	683	858	1,084	1	5	10
sphinx3	1,495	2,645	3,546	3	10	16
hmmer	2,725	3,760	4,391	4	12	21
man	102	265	443	1	5	12
gzip	66	183	365	1	4	11

TABLE III
STATISTICS OF CODE CLONES DETECTED FROM **TWIN-FINDER** WITH THE NUMBER OF ITERATIONS FOR FEEDBACK UNTIL CONVERGE WHERE S IS THE CODE SIMILARITY

We evaluated the effectiveness of **Twin-Finder** to show the optimal results **Twin-Finder** is able to achieve. The code similarity is set as 0.80 with feedback enabled to eliminate false positives until converge (no more false positives can be observed or eliminated) in the first experiment. Table I shows the size of the corresponding percentage of more code clones detected using our approach. As we can see, the results show that **Twin-Finder** is able to detect 393.68% more code clones on average compared to the clone detection without slicing and feedback, with the lowest as 128.80% in *thttpd* and highest up to 913.89% in *gzip*. Note that our approach achieves the best



(a) thttpd (b) links

Fig. 6. The amount of code clones detected in thttpd and links with the number of iterations for feedback until converge after relaxing the code similarity from 0.70 to 1.00

performance in two smaller benchmark *gzip* and *man*. That is because the number of identical code clones is relatively small in both applications (36 in *gzip* and 47 in *man* respectively). While using our approach, we harness the power of program slicing and feedback using formal analysis, which allows us to detect more true code clones.

We further evaluated the coverage of code clones detected in terms of pointer-related code. We measured the total number of pointer-related lines of code (LoC) cross the entire program and the detected LoC using DECKARD and our approach as shown in Table II. It presents the total detected pointer related cloned lines, named as *Domain Specific LoC* (D.S LoC), using our approach. The percentage of D.S LoC ranges from 40.15% to 75.89%, while for DECKARD the number ranges from 9.57% to 32.51%. The results show that it is difficult to directly compare the coverage for different applications, because such results are usually sensitive to: (1) the type of application, such as sphinx3 has intensive pointer access, thus it has the highest clone coverage using our approach; (2) the different configurations may lead to different results(e.g.,different similarity S).

In the second experiment, we relaxed the code similarity threshold from 1.00 to 0.70 to show our approach is capable to detect many more code clones within a flexible user-defined configuration. However, it is reasonable to expect more false positives to occur.

To tackle such false positives issue, we enabled a closed-loop feedback to vector embeddingIn this experiment, we applied our feedback as soon as we observed two code clone samples having different constraints through our clone verification process. We executed several iterations of our feedback until the percentage of false positives converged (no more false positives can be eliminated or observed). Figure 6 presents the number of true code clones detected in thttpd and links from our approach (drawn as the red line in each figure) and the number of iterations for feedback needed to converge (shown as the bar plot in each figure) correspondingly. We also repeated the same experiments with three different code similarities setups in other smaller benchmarks. Table III shows the results. As expected, we are able to detect more true code clones while we reduce the code similarity.

C. Feedback for False Positives Elimination

We analyzed the number of false positives that could be eliminated by our approach. Here, we chose bzip, thttpd and Links as representative applications to show the results. Figure 7 presents the accumulated percentage of false positives eliminated by **Twin-Finder** in each iteration with Code Similarity

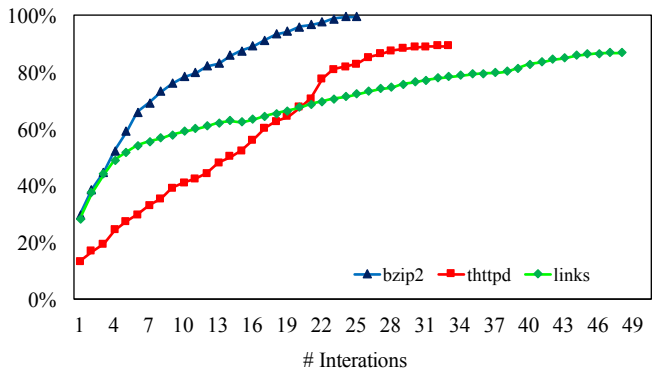


Fig. 7. Accumulated percentage of false positives eliminated by **Twin-Finder** with code similarity set to 0.70

set to 0.7. Here, we are able to eliminate 99.32%, 89.0%, and 86.74% of false positives in bzip2, thttpd and Links respectively. The results show our feedback mechanism can effectively remove the majority of false positives admitted from code clone detection. The performance of our feedback is sensitive to different programs due to different program behaviors and program size. Finally, our feedback may not be able to remove 100% of false positives, that is because there are several special cases that we cannot remove them using current implementation, such as multiple branches or indirect memory access with the value of array index derived from another pointer.

V. Related Work

Code clone detection. Traditional text-based or tree-based approaches are still not sufficient to detect semantics-similar code clones. Thus, learning-based approaches have been developed over the past three years. White et al. [11] first proposes deep neural network (DNN) based code clone detection in source code. But still, they are not able to detect non-contiguous and intertwined code clones. Komondoor et al. [8] also make the use of program slicing and dependence analysis to find non-contiguous and intertwined code clones. But they are trying to find isomorphic subgraphs from program dependency graph in order to identify code clones, where the computing of graph comparison is more expensive. And they do not apply a variant code similarity metric and formal analysis.

Learning-based approach for code analysis. Prior work have studied bug/vulnerabilities using learning based approaches [18], [17], [14], [12], [13]. StatSym [19] proposes frameworks combining statistical and formal analysis for vulnerable path discovery. SIMBER [15] proposes a statistical inference framework to eliminate redundant bound checks and improve the performance of applications without sacrificing security. In this paper, we develop an integrated framework that harnesses the effectiveness of code clone detection and formal analysis techniques on source code at scale.

VI. Conclusion

In this paper, we presented a novel framework, **Twin-Finder**, a pointer-related code clone detector for source code, that can automatically identify related codes from large code bases and perform code clone detection. We evaluated our approach using real-world applications, such as SPEC 2006 benchmark suite. Our results show **Twin-Finder** is able to detect up to 9× more code clones comparing to conventional code clone detection approaches and remove an average 91.69% false positives.

Acknowledgments

This work was supported by the US Office of Naval Research (ONR) under Awards N00014-15-1-2210 and N00014-17-1-2786. Any opinions, findings, conclusions, or recommendations expressed in this article are those of the authors, and do not necessarily reflect those of ONR.

References

[1] "SPEC CPU 2006," https://www.spec.org/cpu2006/, 2006.
[2] ACME Lab, "Thttpd," http://www.acme.com/software/thttpd/.
[3] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.
[4] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the twentieth annual symposium on Computational geometry*. ACM, 2004, pp. 253–262.
[5] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 96–105.
[6] C. J. Kapser and M. W. Godfrey, ""Cloning considered harmful" considered harmful: patterns of cloning in software," *Empirical Software Engineering*, 2008.
[7] H. Kim, Y. Jung, S. Kim, and K. Yi, "Mecc: memory comparison-based clone detector," in *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 301–310.
[8] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *International Static Analysis Symposium*. Springer, 2001, pp. 40–56.
[9] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "Bugbench: Benchmarks for evaluating bug detection tools," in *Workshop on the evaluation of software defect detection tools*, vol. 5, 2005.
[10] Twibright Labs, "Links," http://links.twibright.com.
[11] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 87–98.
[12] H. Xue, "Learn2reason: Joint statistical and formal learning approach to improve the robustness and time-to-solution for software security," Ph.D. dissertation, The George Washington University, 2020.
[13] H. Xue, Y. Chen, G. Venkataramani, and T. Lan, "Hecate: Automated customization of program and communication features to reduce attack surfaces," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2019, pp. 305–319.
[14] H. Xue, Y. Chen, G. Venkataramani, T. Lan, G. Jin, and J. Li, "Morph: Enhancing system security through interactive customization of application and communication protocol features," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2315–2317.
[15] H. Xue, Y. Chen, F. Yao, Y. Li, T. Lan, and G. Venkataramani, "Simber: Eliminating redundant memory bound checks via statistical inference," in *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 2017, pp. 413–426.
[16] H. Xue, S. Sun, G. Venkataramani, and T. Lan, "Machine learning-based analysis of program binaries: A comprehensive study," *IEEE Access*, vol. 7, pp. 65 889–65 912, 2019.
[17] H. Xue, G. Venkataramani, and T. Lan, "Clone-hunter: accelerated bound checks elimination via binary code clone detection," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2018, pp. 11–19.
[18] ——, "Clone-slicer: Detecting domain specific binary code clones through program slicing," in *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation*. ACM, 2018, pp. 27–33.
[19] F. Yao, Y. Li, Y. Chen, H. Xue, T. Lan, and G. Venkataramani, "Statsym: vulnerable path discovery through statistics-guided symbolic execution," in *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on*. IEEE, 2017, pp. 109–120.
[20] Y. Zheng, X. Zhang, and V. Ganesh, "Z3-str: A z3-based string solver for web application analysis," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 114–124.