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

Hongfa Xue, Guru Venkataramani, Tian Lan

***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 *Twin-Finder* that can swiftly identify up 9× more code clones comparing to conventional code clone detection approaches. We conduct security analysis for memory safety using real-world applications Links version 2.14 and libreOffice-6.0.0.1. *Twin-Finder* is able to find 6 unreported bugs in Links version 2.14 and one public patched bug in libreOffice-6.0.0.1.**

***Index Terms*—Code Clone Detection, Machine Learning, Memory Safety**

I. Introduction

With rapid rise in software sizes and complexity, analyzing and fixing bugs in large scale applications is becoming increasingly critical. Debugging and patching security flaws at scale are practically required in software development and engineering. Fortunately, similar code fragments are common in large code bases [18], [24], [28], [44]. 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 work that use token subsequence matching, tree or control flow based graph analysis [6], [23], [22] have shown good performance in detecting *text-based similar code clones* (e.g. copy and paste code fragments). 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 clustering algorithms, improves the previous string-matching based clone detections by introducing a code similarity measurement and transferring the code into intermediate representations (e.g. feature vectors) to detect more code clones [9], [8], [13], [14]. However, this may cause a considerable amount of false positives due to a smaller code similarity threshold. Therefore, more effective code clone detection approaches are necessary to better aid software development process.

In this paper, we introduce a novel clone detection approach,**Twin-Finder**, that is designed for better security analysis in large scale systems. 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. As an example, since pointers and pointer-related operations widely exist in real-world applications and often cause security bugs [11], [35], [15], detecting such pointer-specific code clones are of great significance. We note that a similar approach could be adopted to identify domains relating to any data-flow or control-flow specific code.

In this work, we design and demonstrate our framework for pointer-related code clone detection. We first perform pointer dependency analysis using lightweight tainting to traverse the program control flow graph and find pointer-related operations that can affect change buffer bounds. Then, we leverage both backward and forward program slicing to remove pointer irrelevant codes and isolate pointers in order to find non-contiguous and possibly intertwined pointer-related code clone samples. By doing so, we are able to improve the number of code clones detected, as well as the coverage of code base with respect to finding relevant code clones that ultimately helps with rapid security analysis. To facilitate higher code coverage, we also explore a wide range of code similarity threshold for the detection process.

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. We use a recursive sampling approach to randomly divide each grouped cluster into smaller ones. We sample each such smaller cluster of code clones and make all the pointer related variables as symbolic variables. We then apply symbolic execution to verify if they are true code clones, as SE is able to symbolic execute and explore all the possible paths to collect memory bound checking conditions. Two code clone samples are determined as true code clone pair if they both share the same memory safety constraints. Moreover, it is highly likely that code clone detection algorithm can still cause false positives. Existing works have reported that the false positives from code clone detection are inevitable [34], [3]. Even though deep learning-based approaches are able to reduce a modest amount of false positives comparing to prior works, human efforts are still needed for further verification and tuning detection algorithms. 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 constraints. 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.

We have implemented a prototype of **Twin-Finder** with two major modules: Domain Specific Slicing and Closed-loop Code Clone Detection. It utilizes several open-source tools and presents a new closed-loop operation with the

assistance of formal analysis. In particular, we use a static code analysis tool, Joern [47] and develop a program slicing framework. We instrument a tree-based code clone detection tool, DECKARD [22], to detect code clones after slicing. We employ a source code symbolic execution tool, KLEE [12], for our clone verification and feedback to vector embedding in previous code clone detection module.

We evaluate the effectiveness of **Twin-Finder** in real-world applications, such as Links [36], thttpd [2]. For code clone detection, we apply **Twin-Finder** to evaluate the number of code clones detected against conventional code clone detection approaches. We further construct security case studies for vulnerability discovery. The results show **Twin-Finder** finds 6 unreported bugs in Links version 2.14 and one public reported bug in libreOffice-6.0.0.1, including 3 memory leaks and 3 Null Dereference vulnerabilities. And 1 of the memory leaks bug is silently patched in the newer version of Links. We further compare the overhead of our clone verification module using symbolic execution and the execution time with pure symbolic execution over entire binary programs to find the bugs.

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 to enable a rapid security analysis.
- **Twin-Finder** leverages program slicing to remove irrelevant codes and isolate analysis targets to find non-contiguous and intertwined code clones so that the detection can pin down just-enough information to adapt to a specific domain.
- **Twin-Finder** deploys formal analysis to perform a closed-loop operation. In particular, NAME introduces a clone verification mechanism to formally verify if tow clone samples are indeed clones and a feedback loop to tune code clone detection algorithm and further reduce the false positives.
- We implement a prototype of **Twin-Finder** using several open-source tools, including Joern, DECKARD, and KLEE. Our evaluation demonstrates that **Twin-Finder**, with the optimal configuration, can detect up to $9\times$ more code clones comparing to conventional code clone detection approaches.
- We conduct case studies of pointer analysis for memory safety using real-world applications We show that using **Twin-Finder** we find 6 unreported bugs in Links version 2.14 and one public patched bug in libreOffice-6.0.0.1.

The rest of this paper is structured as follows: We first list the limitations of existing code clone detection approaches and the key solutions of our approach. We give the overview of **Twin-Finder** in Section III and introduce the designs of **Twin-Finder** along with technical details in Section IV. We evaluate its effectiveness to detect code clones in real-world applications and conduct a case study about security analysis in Section VI. Finally, we discuss the related work and concludes the paper in Section VII and Section VIII respectively.

Similarity	#True Positives	#False Positives	%False Positives
$= 1.00$	1,495	0	0.00%
≥ 0.95	2,016	203	9.15%
≥ 0.90	2,637	394	13.00%
≥ 0.85	3,017	585	16.24%
≥ 0.80	3,526	903	20.75%

TABLE I: Clone statistics of true positives and false positives detected from sphinx3 benchmark using DECKARD

II. Problem Statement and Motivation

A. Code Clone Detection and Challenges

Many software engineering tasks, such as refactoring, understanding code quality, or detecting bugs, require the extraction of syntactically or semantically similar code fragments (usually referred to as *"code clones"*). Generally, there are three code clone types.**Type 1**: Identical code fragments except for variations in identifier names and literal values; **Type 2**: Syntactically similar fragments that differ at the statement level. The fragments have statements added, modified, or removed with respect to each other. **Type 3**: Syntactically dissimilar code fragments that implement the same functionality.

Code clone detection approaches comprise two phases in general: (i) Transfer code into an intermediate representation, such as tree-based clone detection declaring feature vectors to represent code fragments [31]; (ii) Deploy suitable similarity detection algorithms to detect code clones. For instance, clustering algorithms from machine learning are widely used in code clone detection problems [22]. Some existing code clone detection techniques apply simple pattern matching (e.g., token-based code clone detection approach [4], [23], [28]) and leverage a code similarity metric to measure the amount of similarity between two code samples.

Assuming we want to detect code clones in 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. Therefore, we need better alternatives to the current state of the art solutions.

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. Here, we analyzed the true positives and the false positives detected using conventional tree-based code clone detection approach with different code similarity thresholds. We select *sphinx3* as representative applications and the results are shown in Table I. As we can see, relaxing code similarity threshold can benefit detection with more code clone samples. However, the ratio of false positives also increases at the time. If we can eliminate the false positives 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 [22] as motiving examples. First, we give the formal definition of false positive which is defined in Definition 1.

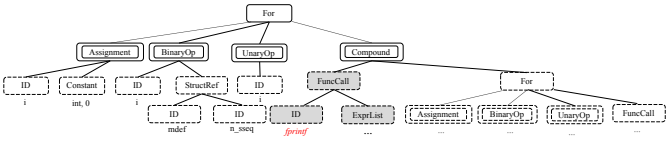
```
1  void dict2pid_dump (...){
2  ...
3  for (i = 0; i < mdef->n_sseq; i++) {
4    fprintf (fp, "%5d ", i);
5    for (j = 0; j < mdef_n_emit_state(mdef); j++)
6      fprintf (fp, "%5d", mdef->sseq[i][j]);
7  ..
8  }
9  ..
10 }
```

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

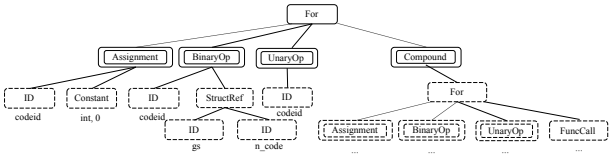
```
1  int32 gc_compute_closest_cw (...){
2  ...
3  for(codeid=0; codeid< gs->n_code ;codeid+=2){
4    for(cid=0;cid<gs->n_featlen ; cid++)
5      fprintf (fp, "%5d", gs->codeword[codeid][cid]);
6  }
7  ...
8  }
9  ...
10 }
```

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

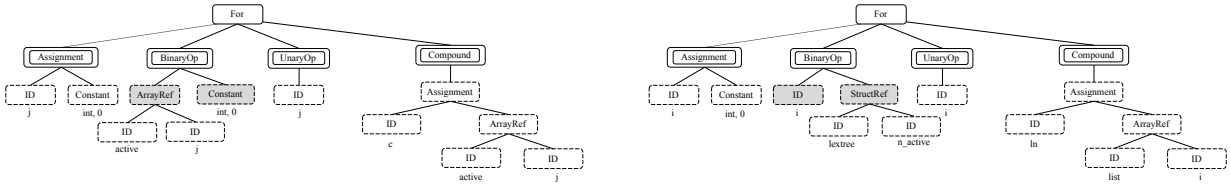
```
1  void fe_spec_magnitude(double *data, int32
       data_len, double *spec, int32 fftsize)
2  {
3  ...
4  IN = (complex *) calloc(fftsize,sizeof(complex));
5  ...
6  for (wrap=0; j<data_len; wrap++,j++) {
7    IN[wrap].r += data[j];
8    IN[wrap].i += 0.0;
9    }
10   ...
11 }
12 ...
13 ...
14 for (j=0; j<fftsize;j++) {
15   IN[j].r = data[j];
16   IN[j].i = 0.0;
17   }
18 ...
19 }
```

Code clone samples of function *sphinx3::fe_spec_magnitude* as pointer $\{IN\}$ are intertwined inside of two different *for* loops of the function

Fig. 3: False positive examples from sphinx3

Definition 1. ***False Positives.*** *In this paper, we define as 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 transferring 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. However, more advanced clone detection approaches have been proposed, which can be summarized into two methods: graph matching-based and



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

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

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

deep neural network (DNN)-based approach. Unfortunately, they still have inevitable drawbacks. First, given two pieces of code which differ in only a few statements but with the similar control flow, in the graph matching-based clone detection, they may be considered as similar, since the majority of the code is identical. On the other hand, current DNN-based clone detection is only used to detect identical code clone (e.g., with code similarity S = 1.0). Thus, if the similarity threshold is set as $S < 1.0$, the outputs will be similar to traditional tree-based/token-based approach. It is clear to see that the first code sample has an extra function call *fprintf* comparing to the second code sample. If we relax the code similarity threshold, these two code samples are identified 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.

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

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

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 two false-positive examples detected in sphinx3 from SPEC2006 benchmark. For the first example (showing on the left-hand side of the figure), two *for* loops are identified as code clones (line4-5 and line 15-16) 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. Similarly, the second example (showing on the right-hand side) are sharing a similar code structure but differs only in identifier names. Thus, they can also be identified as code clones. Assuming the target pointers for analysis are $active$ and $list$ in the first example, we first to 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. In the second example, the same dependency analysis procedure is deployed. Variables $\{wrap, j, data\_len\}$ are identified as pointer related variables in the first *for* loop (line6-9) and $\{j, fftsize\}$ are related to second *for* loop (line14-17), they are also false positives which are similar to the first example. One of the reasons to cause false positives in both cases are relaxed code similarity threshold to seek non-identical code clones.

To formally verify if two code clones are true positives or false positives, symbolic execution can be applied to obtain memory safety conditions for further condition comparison. First, all pointer related variables of target pointers are made as symbolic variables. Symbolic execution can execute for each pointer dereference and generate array bounds safety conditions. To further eliminate false positives, in this paper, we propose a feedback loop to clone detection module through formal analysis. Once a false positive occur, we compare the ASTs representing two clone samples to find the different nodes and add numerical weight to those nodes so that we can recalculate the code similarity between two trees to reduce the false positives admitted from code clone detection. For example, we note that the different nodes are {**ID, StrucRef**} and {**ArrayRef, Constant**} for the example showing in Figure 4 respectively. Then we can simply add weight to each of those nodes. With a fixed code similarity, those two code samples will be eliminated in the future.

III. Approach Overview

In this section, we give an overview of our framework. Two main components of **Twin-Finder** are shown in Figure 5, namely Domain Specific Slicing and Closed-loop Code Clone Detection.

Domain Specific Program Slicing: We use tainting as a flexible mechanism to identify user-defined domains for further analysis. For a demonstration in this paper, we will use pointer analysis as the domain of interest. For a given program source code, Twin-Finder first generates dependency graph based on data and control information and uses a lightweight tainting approach to traverse the graph and find pointer related variables (such as array index). We then utilize program slicing to isolate pointers and the corresponding related statements.

Closed-loop Code Clone Detection: After we generate pointer isolated code (containing pointers and their related code by including all of the variables and statements that affect them), code clone detection algorithm is applied to identify code clone samples. Toward this, we first generate Abstract Syntax Trees (AST) for each code fragment and transform

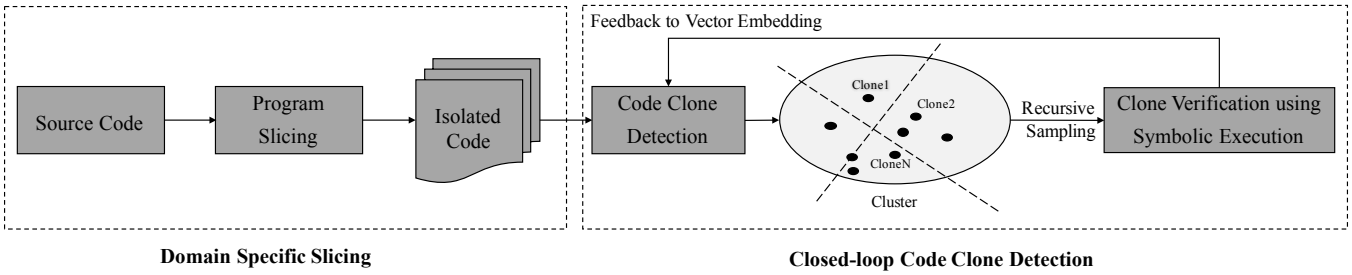


Fig. 5: Approach Overview

such ASTs into feature vectors, embed them into vector space and use clustering algorithm from machine learning to find code clones. Note that we also use various code similarity thresholds to further increase the number of detected code clones(Section IV-B). A key distinguishing feature of our approach compared to prior work lies in improving the robustness of code clones detected through formal methods for verification. In particular, we use symbolic execution to verify whether the code clones grouped in the same clusters are indeed clones with respect to memory safety (that is, the pointer access is within legal array bounds). We define two clones are true clones only if they have the same array bound constraints (formed using pointer-affecting variables in the program code). We propose a recursive sampling mechanism for the clone verification and the process is performed as follows: In order to improve the analysis coverage among all the code clone pairs, we first randomly divide each cluster into smaller clusters. We then sample each such smaller cluster of code clones and apply symbolic execution to verify whether two clone samples share the same memory safety constraints derived using symbolic executors. We note that code clones within the same cluster may potentially have different constraints stemming from variables that affect their values. We consider such clones as *false positives* introduced by code clone detection algorithm. If the selected clone samples are falsified by formal analysis, we enable a formal feedback mechanism to tune the feature vector weights accordingly to eliminate such false positives from occurring again. Section IV-C describes our design of this module.

IV. System Design

In this section, we present details of our Twin-finder and show how our system is designed.

A. Domain Specific Slicing

For pointer analysis, we aim to analyze each pointer in the program to ensure there is no issue like memory violation. Thus, 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*. In this paper, we use dependency analysis to find such pointer-related variables for each pointer on a function-level granularity. Then, we deploy both forward and backward program slicing to select related statements containing pointer and 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 between 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 four 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). In particular, we use a program parser ANTLR[32] and a static code analysis tool Joren [47] to analyze program syntax. 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.** After we obtain all the target pointers and their corresponding pointer-related variables, 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. For example, a line of statement $v_i = x$

will be kept, but $y = v_i$ will be removed since it cannot change the value of v_i. 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. After performing program slicing, we are able to isolate one single function into several pointer separated functions. For instance, if there are 10 pointers in one function, then there should be 10 pointer isolated functions derived from this function. Note that it is possible that one statement involves multiple pointers, this type of statements will be selected in all the involved pointers. In additions, we also need to preserve the locations (e.g. line of code) of any selected statements in the original source code for further analysis.

B. Code Clone Detection

Twin-Finder leverages a tree-based code clone detection approach, which is originally proposed by Jiang et al. [22]. It produces the Abstract Syntax Tree (AST) representation of the source program to detect code clones by comparing subtrees in ASTs with a specific similarity metric. AST is commonly used tree representation by compilers to abstract syntactic structure of the code and to analyze the dependencies between variables and statements. The source code can be parsed by using the static code analysis mentioned in Section IV-A and generate AST correspondingly. Here, we adopt the notions of code similarity, feature vectors and other related definitions from previous works [10], [22]. 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 weighted code similarity S between them is defined as:

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

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. More details related to AST nodes types can be found in [8].*

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. *Example.* The feature vector for the code fragment of function *sphinx3::mgau_eval*, mentioned in Section II-B, is $< 7, 2, 2, 2, 0, 1, 1, 1, 1 >$ where the ordered dimensions of vectors are occurrence counts of the relevant nodes: **ID, Constant, ArrayRef, Assignment, StrucRef, BinaryOp, UnaryOp, Compound,** and **For**.

2) Clone Detection: Given a group of feature vectors, we utilize Locality Sensitive Hashing (LSH) [16] and near-neighbor querying algorithm based on the euclidean distance between two vectors to cluster a vector group, where LSH can hash two similar vectors to the same hash value and helps near-neighbor querying algorithm to form clusters [22], [20]. 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 $E([V_i; V_j])$ and hamming distance $H([V_i; V_j])$ between V_i and V_j are calculated as following:

$$E([V_i; V_j]) = ||V_i - V_j||_2^2 \tag{4}$$

$$H([V_i; V_j]) = ||V_i - V_j||_1 \tag{5}$$

The threshold used for clustering can be approximated using the euclidean distance and hamming distance between two feature vectors for two ASTs T_1 and T_2 as following:

$$E([V_i; V_j]) \geq \sqrt{H([V_i; V_j])} \approx \sqrt{L + R} \tag{6}$$

Based on the definition from Equation 2, we can derive that $\sqrt{L + R} = \sqrt{2(1 - S) \times (|T_1| + |T_2|)}$, where $(|T_1| + |T_2|) \geq 2 \times min(S(C_i), S(C_j))$. Then, the threshold for the clustering procedure is defined as:

$$T = \sqrt{2(1 - S) \times min(S(C_i), S(C_j))} \tag{7}$$

Then, given a feature vector group V, the threshold can be simplified as $2(1-S) \times min_{v \in V} \in S(v)$, where we use vector sizes to approximate tree sizes. The S is the code similarity metric defined from Equation 2. 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.

There are three phases of clone verification: (1) Recursive sampling code clones in clusters;(2) Deploy symbolic execution and constraints solving for clone verification; (3) A feedback mechanism to vector embedding in previous code clone detection module to improve the correctness of clustering algorithm and eliminate false positives.

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 partial 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 $AC(b_1) = \{i < length(mdef- > sseq)\}$. Similary, 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. To deal with possibly incomplete program state while performing partial symbolic execution, we only make the pointer-related variables in such code fragment as symbolic variables. 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. In specific, the current state-of-art symbolic execution approaches, such as KLEE [12], use SMT-Lib string constraints format with BitVector theory [7], [19]. The operations in BitVector theory are modeling array and variables on bit-vectors instead of integer values. For example, $(declare-fun\ a()\ (Array(\_BitVec32)(\_BitVec8)))$ stands for an array with symbolic variable name a, total length as 32 bits and return value as 8 bit long. Thus, this array has $32/8 = 4$ elements (Here, we omit the details of BitVector theory as this is not the focus of this verification process.)

The steps of this verification process are summarized as follows:

- **Variables Matching**: 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. For example, a linear inequality $(x - n < 0) \wedge (x - z > 0)$, after simplification, we can get $(z < x < n)$.
- **Equivalence Checking**: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

```
Input:: Code Clone Samples C_i, C_j
Corresponding AST sub-trees: S_i, S_j
Corresponding Feature Vectors: V_i, V_j
Current Code similarity threshold: S
Longest Common Subsequence function: LCS ()
Output:: Optimized Feature vectors: O_i, O_j
Initialization:
O_i, O_j = V_i, V_j
D = LCS(S_i, S_j)
if C_i and C_j share same constraints then
    S_i = RemoveSubtrees(S_i − D)
    S_j = RemoveSubtrees(S_j − D)
    O_{n∈{i;j}} = Vectornize(S_{n∈{i;j}});
else
    T=[]
    Uncommon_Subtrees = (S_i − D) + (S_j − D)
    T.append(Uncommon_Subtrees)
    for t in T do
        if EuclideanDistance(O_i, O_j) < S then
            break;
        t = d.index
        O_{n∈{i;j}}[t] = O_{n∈{i;j}}[t] ∗ δ; where δ > 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 pointson 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 IV-C2.

The general idea of our feedback is that we analyze the difference between two ASTs by comparing two trees and find the differences in between. Then we add numerical weights to the feature vectors of two code clones to either increase or decrease the distance between them based on the outputs from the clone verification step. Once the weight is added, we re-execute the clustering algorithm in code clone detection module over the same code similarity threshold configuration. Note that this procedure can be executed in many iterations as long as we observe false positives from clone verification step. Furthermore, we can expect that such false positives are eliminated due to unsatisfied vector distance and out of cluster boundary.

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 details. 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. In terms of the evolution of the weight adjustment,each dimension in the feature vectors represents a specific type of AST nodes and is the occurrences of this node type. Thus, we iterate the list and we trace back to the vector using the vector index to adjust by multiplying the weight 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 eucildean 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. As we have described in Section II-B, these two trees share a common tree pattern but with 2 different nodes (showing in shady color) out of 17 total nodes. 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 6, 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 eucildean 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.

It is also worth mentioning that our feedback algorithm has enabled a closed-loop learning-based operation to improve the scalability of our pointer-related code clone detection framework. Because this method adds benefits from formal analysis and can significantly reduce the false positives without human efforts involved. Here, we use pointer analysis as an example to explain our framework. In addition, our feedback algorithm can be adjusted to different domains with user-defined policies.

V. Implementation

This section discusses our implementation of **Twin-Finder** and how we integrate the tools we used.

Program Slicing: We instrument a static code analysis tool, Joern [47], for our program slicing module. Joern is able to store code property graphs (like ASTs) in a Neo4J graph database [38], here we call it AST database, for user to write their own scripts to do static code analysis. We develop a python script to build ASTs for each function and construct dependency graphs. After, we store them into Neo4J graph database for further analysis. As Joern cannot store the source code location information, such as which lines these statements is from in the source code. We instrument Joern to include additional information for a certain statement using a C++ script, including file path along with code line number, so that we can trace back to source code after we perform static program slicing to isolate original source code into pointer isolated functions.

Code Clone Detection: DECKARD [22], a static Code Clone Detection tool, is used for code clone detection in **Twin-Finder**. DECKARD is a tree-based code clones detection tool that computes certain characteristic vectors within code parse trees and then clustering these vectors depending on their Euclidean distances. We instrumented DECKARD interfaced with our program slicing module to automate the clone detection process.

Clone Verification: We instrument a source code symbolic execution tool, KLEE [12] and SMT solver Z3 [51] for our clone verification module. We first develop a python script to automatically add codes into the pointer isolated code fragments and make pointer-related variables symbolic using KLEE provide library function. We then deploy the symbolic executor in KLEE for a target location to start performing symbolic execution in the source code, beginning with the

starting line of code and execute till the ending line of code in the code fragment. Finally, we implemented our feedback in also python based on the algorithm proposed in Section IV-D.

VI. Evaluation

This section presents a detailed evaluation results of **Twin-Finder** against a tree-based code clone detection tool DECKARD [22] in terms of code clone detection, and conduct several case studies for applications security analysis.

A. Experiment Setup

We performed empirical experiments on **Twin-Finder**Ẇe selected 7 different benchmarks from real-world applications: bzip2, hmmer and sphinx3 from SPEC2006 benchmark suite [1]; man and gzip from Bugbench [30]; thttpd-2.23beat1 [2], a well-known lightweight sever and a lightweight browser links-2.14 [36]. All experiments are performed on a 2.54 GHz Intel Xeon(R) CPU E5540 8-core server with 12 GByte of main memory. The operating system is ubuntu 14.04 LTS.

To configure DECKARD, we used the parameter settings proposed by Jiang et al. [22], setting minimum token number (minT) as 20, stride to infinite, and code similarity is set between 0.70 and 1.0.

B. Code Clones Detection

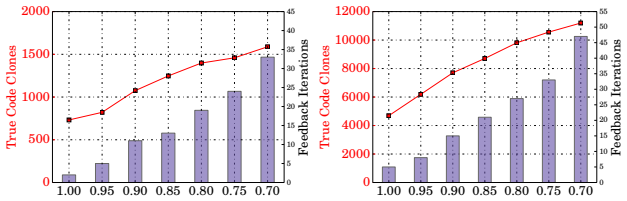
Benchmark	Program Size (LoC)	#Code clones without slicing and feedback	#Code clones Our approach	% Code clones
bzip2	5,904	432	1,084	150.92%
sphinx3	13,207	1,047	3,546	238.68%
hmmer	20,721	1,238	4,391	254.68%
thttpd	7,956	611	1,398	128.80%
gzip	5,225	36	365	913.89%
man	3,028	47	443	842.55%
links	178,441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

TABLE II: Comparison of number of code clones detected before and after using our approach

Benchmark	Pointer related Code	Clone Detection w/ DECKARD		Clone Detection w/ Our Approach	
	LoC	# 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 III: Comparison of code clone coverage between DECKARD and 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.



(a) thttpd (b) links

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

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 IV: 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** are 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 II 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 in 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 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.

Furthermore, we add an additional experiment to address the clone coverage. The goal for clone coverage is, with our optimal configuration, what fraction of a program is detected as cloned code. In this case, we only evaluated the coverage of code clones detected in terms of pointer-related code. We measured the total number of pointer-related code lines cross the entire program and the detected clone lines using DECKARD and our approach as shown in Table III. 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 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, since here we set up code similarity as 0.80. However, this experiment is to show that there is a considerable amount of code clones in large code bases in general and our approach can effectively detect such clones and outperforms previous approaches.

In the second experiment, we relaxed the code similarity threshold from 0.70 to 1.00 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 while we are using smaller code similarity. Moreover, we implemented our code clone detection based on DECKARD, which is a syntax tree-based tool and may report semantically different but syntactically similar code as clones causing more false positives. Note that false negatives occur if tow clone samples have different constraints but are actually the same expression after being solved by the constraint solver. However, false negatives only result in actually true clones being missed by **Twin-Finder** and are not critical in security perspective. Thus, we do not evaluate **Twin-Finder** for false negatives in our study.

To tackle such false positives issue, we enabled a closed-loop feedback to vector embedding as mentioned in the previous section. Thus, we analyzed the effectiveness of our feedback mechanism in terms of eliminating the false positives. In this experiment, we applied our feedback as soon as we observed two code clone samples having different constraints obtained from symbolic execution 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 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 IV shows the results. As expected, it takes more iterations for the feedback to converge with smaller code similarity among all benchmarks, and we are still able to detect more true code clones while we reduce the code similarity. However, the results show there is no significant improvement in terms of the number of true code clones increased after code similarity is set as smaller than 0.80. As mentioned in previous section, the code similarity is defined as $S(T_1, T_2) = \frac{2S}{2S+L+R}$, where S is the number of shared AST nodes in T_1 and T_2, L and R are the different nodes in two code clone samples. At least 20% of the AST nodes are different while the code similarity equal to 0.80.

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 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

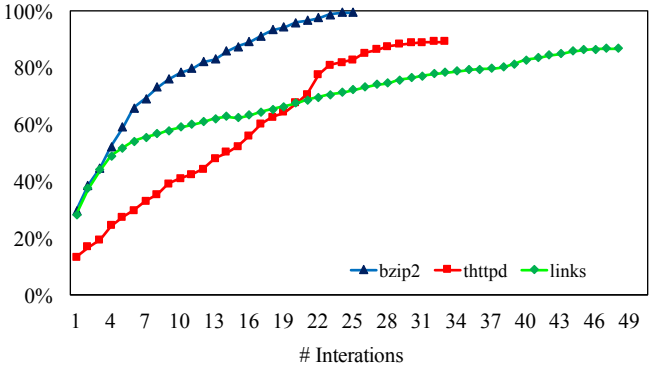


Fig. 7: Accumulated percentage of false positives eliminated by **Twin-Finder** with code similarity set to 0.70

to different programs due to different program behaviors and program size. As the results show, more feedback iterations are needed for larger program in general (e.g. 26 iterations for bzip2 to converge while 48 iterations for Links, as Links is much larger than bzip2). On the other hand, the number of iterations can also be affected by our clone verification module since we use random sampling approach. Based on the experiment results, we cannot normalize a common removal ratio pattern cross different programs. For instance, 29.83% of false positives can be eliminated at the first iteration for bzip2, the number is only 13.35% for thttpd instead. 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.

D. Bug findings

One benefit of our approach is to use a clone-based approach to enable a rapid security analysis. In this experiment, we use **Twin-Finder** to detect potential vulnerabilities existing in the applications. We use Links version 2.14 and LibreOffice version 6.0.0.1 as representative benchmarks. In particular, we discovered 6 unique and unreported bugs in Links, including 3 memory leaks and 3 null dereference vulnerabilities. five out of six of the bugs have not been found before, and one of the memory leaks bug has been silently patched in the newer version of Links.

Table V shows the details of these bugs found by our method. Here we show three types of bug examples, null dereference bugs, memory leak and buffer overflow.

1) Links Case Study: In the first case study, we employ **Twin-Finder** to uncover vulnerabilities in Links, a lightweight browser. The results show **Twin-Finder** finds 6 unreported bugs in Links version 2.14, including 3 memory leaks and 3 Null dereference vulnerabilities. And 1 of the memory leaks bug is silently patched in the newer version of Links.

As an example, let us consider the function *get_language_from_lang* shown in Figure 8. This function is implemented as setting language from local serves. This function provides an illustrative example because the programmer confirms that the $stracpy$ requires validation in the comment on line 5 The $stracpy$ function in line 4 is

Bug Type	Source File	Function Name	Pointer Name	Bug Report	Exploitation
Null Dereference	Links-1.4/language.c	get_language_from_lang	lang	Not Reported	All three cases use memory allocation functions, which can be return
Null Dereference	Links-1.4/language.c	get_language_from_lang	p	Not Reported	NULL to indicate an error status. When this error condition is not checked, a NULL pointer dereference can occur.
Null Dereference	Links-1.4/connect.c	make_connection	host	Not Reported	This function is being called in a for loop to construct network connection, which can potentially be frequently called and overflow the memory space.
Memory Leak	Links-1.4/ftp.c	ftp_logged	rb	Not Reported	All those three functions use dynamic allocations, however never free after.
Memory Leak	Links-1.4/bfu.c	do_tab_compl	items->text	Silently patched	
Memory Leak	Links-1.4/terminal.c	add_empty_window	ewd	Not Reported	
Buffer Overflow	libreoffice-6.0.0.1/sw/source/filter /ww8/ww8toolbar.cxx	SwCTBWrapper::Read	rCustomizations	Publicly patched	This is a heap buffer overflow, which has been reported by CVE-2018-10120

TABLE V: Using our approach to test Links-1.4 and libreoffice-6.0.0.1

```
1  int get_language_from_lang(unsigned char *lang)
       {
2  unsigned char *p;
3  int i;
4  lang = stracpy(lang);
5  //Uncheck the memory allocation
6  lang[strcspn(cast_const_char lang, ".@")] = 0;
7  if (!casestrcmp(lang, cast_uchar "nn_NO"))
8    strcpy(cast_char lang, "no");
9  ...
10 search_again:
11 for (i = 0; i < n_languages(); i++) {
12   p = cast_uchar translations[i].t[
       T_ACCEPT_LANGUAGE].name;
13   if (!p)
14     continue;
15   p = stracpy(p);
16 //Uncheck the memory allocation
17   p[strcspn(cast_const_char p, ",;")] = 0;
18   if (!casestrcmp(lang, p)) {
19     mem_free(p);
20     mem_free(lang);
21     return i;
22   }
23   mem_free(p);
24 }
25 ...
26 mem_free(lang);
27 return -1;
28 }
```

Fig. 8: Null Dereference bugs in function *get_language_from_lang* of the lightweight browser Links

```
1  bool SwCTBWrapper::Read( SvStream& rS )
2  {
3  ...
4  if (cCust)
5  {
6    ...
7    for (sal_uInt16 index = 0; index < cCust; ++
       index)
8    {
9      Customization aCust( this );
10     if ( !aCust.Read( rS ) )
11       return false;
12     rCustomizations.push_back( aCust );
13   }
14 }
15 ...
16 std::vector< sal_Int16 >::iterator it_end =
       dropDownMenuIndices.end();
17 for ( std::vector< sal_Int16 >::iterator it =
18 dropDownMenuIndices.begin(); it != it_end; ++it
       )
19 {
20   rCustomizations[ *it ].bIsDroppedMenuTB =
       true;
21 }
22 return rS.good();
23 }
```

Fig. 9: Source Code from function *Links::SwCTBWrapper::Read* where a buffer overflow bug via pointer $rCustomizations$

implemented as dynamic memory allocation for a pointer. The bug arises when the code fails to allocate memory to pointer $lang$ using $stracpy$ function and return $NULL$ to pointer $lang$. Thus, there is a potential null pointer dereference in line 6.

After we deploy program slicing and code clone detection, we are also able to identify the same bug with the assistance of symbolic execution rapidly for pointer p in line 15, as two code snippets are identified code clones (line 4-6 and line 15-17). Similarly, pointer p is unchecked after memory allocation, which results in the same vulnerability existing in the codes. This example shows the advantage of our approach combining program slicing and code clone detection for vulnerability discovery.

2) LibreOffice Case Study: In the second case study, LibreOffice is an open source office tool, which is written in multiple programming languages including C/C++ and Java. Currently, our approach is working to C/C++ code only. Thus, we only deployed our approach on the C/C++ files in LibreOffice. Our approach was able to identify a heap-based buffer overflow bug in function *Links::SwCTBWrapper::Read*. Figure 9 shows the original source code. **Twin-Finder** identified a group of code clones of code snippets from line 17-21 in the same cluster. Our feedback mechanism eliminated the other code clones as false positives after 16 iterations.

This function is used to read a crafted document containing a Microsoft Word record (named as a structural array $rCustomizations$ in the source code) from beginning to the end. The size of structural array $rCustomizations$ is defined as $static\_cast < sal\_Int16 > (rCustomizations.size())$. However, the *for* loop in line 17, it does not do a propel bound check of a customizations array index. The value of $*it$ could be negative or larger than the size of $rCustomizations$. When our approach deploys partial symbolic execution for this for loop, it will yield potential buffer overflow error [1].

VII. Related Work

Related works including code clone detection and program slicing have been discussed closely throughout the paper. In

[1]However, after we started our research, this bug has been found earlier of 2018 and public patched in the newer version of LibreOffice. More details about this bug can be found in the report CVE-2018-10120 [29]

this section, we summarize some additional related work. We focus on existing static code analysis and code clone detection approaches. Other approaches for vulnerability discovery will be also discussed in this section.

Code clone detection. Different approaches for code clone detection have been proposed. Recall that detection techniques generally can be classified into several categories. First, text-based or simple string matching based techniques [17], [5], [6] apply slight program transformations and apply a single code similarity measurement by comparing sequences of text. Such text-based techniques are limited in the scalability in large code bases and only finding exact match code clone pairs. Second, tree or token-based clone detections [26], [37], [9], [50] are proposed by parsing program into tokens or generate abstract syntax trees representation of the source program. However, above 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. [39] first proposes deep neural network (DNN) based code clone detection in source code. Similarly, Gemini [40] uses DNN to detect cross-platform code clones in binaries. But still, they are not able to detect non-contiguous and intertwined code clones. Komondoor et al. [25] 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, which 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 vulnerability discovery. Prior work have studied bug/vulnerabilities using learning based approaches [21], [33], [46], [45], [42], [41]. StatSym [49] and SARRE [27] propose frameworks combining statistical and formal analysis for vulnerable path discovery. SIMBER [43] proposes a statistical inference framework to eliminate redundant bound checks and improve the performance of applications without sacrificing security. Another line of work use Natural Language Processing and machine learning to bug detection. For example, Chucky [48] uses context-based Natural Language Processing to detect missing check vulnerability. These techniques, often transfer code into intermediate representation and then rely on static code analysis to find bugs. In this paper, we develop an integrated framework that harness the effectiveness of code clone detection and formal analysis techniques for a rapid security analysis on source code at scale. In contrast to pure formal analysis, such as symbolic execution, we are able to achieve a significant speedup to find vulnerabilities.

VIII. 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 to enable a rapid security analysis. 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. We conduct security case studies for memory safety. In particular, we show that using **Twin-Finder** we find 6 unreported bugs in Links version 2.14 and one public patched bug in libreOffice-6.0.0.1.

References

[1] "SPEC CPU 2006," https://www.spec.org/cpu2006/, 2006.

[2] ACME Lab, "Thttpd," http://www.acme.com/software/thttpd/.

[3] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *arXiv preprint arXiv:1709.06182*, 2017.

[4] B. S. Baker, "A program for identifying duplicated code," *Computing Science and Statistics*, pp. 49–49, 1993.

[5] ——, "On finding duplication and near-duplication in large software systems," in *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*. IEEE, 1995, pp. 86–95.

[6] ——, "Parameterized duplication in strings: Algorithms and an application to software maintenance," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1343–1362, 1997.

[7] C. Barrett, A. Stump, C. Tinelli *et al.*, "The smt-lib standard: Version 2.0," in *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, vol. 13, 2010, p. 14.

[8] H. A. Basit and S. Jarzabek, "Detecting higher-level similarity patterns in programs," in *ACM Sigsoft Software engineering notes*, vol. 30, no. 5. ACM, 2005, pp. 156–165.

[9] I. D. Baxter, C. Pidgeon, and M. Mehlich, "Dms/spl reg: program transformations for practical scalable software evolution," in *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*. IEEE, 2004, pp. 625–634.

[10] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Software Maintenance, 1998. Proceedings., International Conference on*. IEEE, 1998, pp. 368–377.

[11] J. Caballero, G. Grieco, M. Marron, and A. Nappa, "Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 2012, pp. 133–143.

[12] 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.

[13] Y. Chen, T. Lan, and G. Venkataramani, "Damgate: dynamic adaptive multi-feature gating in program binaries," in *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*. ACM, 2017, pp. 23–29.

[14] Y. Chen, S. Sun, T. Lan, and G. Venkataramani, "Toss: Tailoring online server systems through binary feature customization," in *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation*. ACM, 2018, pp. 1–7.

[15] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi, "Losing control: On the effectiveness of control-flow integrity under stack attacks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 952–963.

[16] 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.

[17] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*. IEEE, 1999, pp. 109–118.

[18] M. Gabel and Z. Su, "A study of the uniqueness of source code," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2010, pp. 147–156.

[19] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *International Conference on Computer Aided Verification*. Springer, 2007, pp. 519–531.

[20] A. Gionis, P. Indyk, R. Motwani *et al.*, "Similarity search in high dimensions via hashing," in *Vldb*, vol. 99, no. 6, 1999, pp. 518–529.

[21] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Binary code clone detection across architectures and compiling configurations," in *Program Comprehension (ICPC), 2017 IEEE/ACM 25th International Conference on*. IEEE, 2017, pp. 88–98.

[22] 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.

[23] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[24] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 187–196.
[25] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *International Static Analysis Symposium*. Springer, 2001, pp. 40–56.
[26] K. A. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein, "Pattern matching for clone and concept detection," *Automated Software Engineering*, vol. 3, no. 1-2, pp. 77–108, 1996.
[27] Y. Li, F. Yao, T. Lan, and G. Venkataramani, "Sarre: semantics-aware rule recommendation and enforcement for event paths on android," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 12, pp. 2748–2762, 2016.
[28] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on software Engineering*, vol. 32, no. 3, pp. 176–192, 2006.
[29] LibreOffice, "Cve-2018-10120," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-10120, 2018.
[30] 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.
[31] N. A. Milea, L. Jiang, and S.-C. Khoo, "Vector abstraction and concretization for scalable detection of refactorings," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 86–97.
[32] T. J. Parr and R. W. Quong, "Antlr: A predicated-ll (k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.
[33] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 709–724.
[34] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 2016, pp. 1157–1168.
[35] F. J. Serna, "The info leak era on software exploitation," *Black Hat USA*, 2012.
[36] Twibright Labs, "Links," http://links.twibright.com.
[37] V. Wahler, D. Seipel, J. Wolff, and G. Fischer, "Clone detection in source code by frequent itemset techniques," in *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*. IEEE, 2004, pp. 128–135.
[38] J. Webber, "A programmatic introduction to neo4j," in *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*. ACM, 2012, pp. 217–218.
[39] 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.
[40] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 363–376.
[41] 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.
[42] 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*. ACM, 2018, pp. 2315–2317.
[43] 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.
[44] 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.
[45] 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*. ACM, 2018, pp. 11–19.
[46] ——, "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.
[47] F. Yamaguchi, "Joern: A Robust Code Analysis Platform for C/C++," http://www.mlsec.org/joern/, 2016.
[48] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: Exposing missing checks in source code for vulnerability discovery," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 499–510.
[49] 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.
[50] K. Zhang, M. Wang, X. Cong, F. Huang, H. Xue, L. Li, and Z. Gao, "Personal attributes extraction based on the combination of trigger words, dictionary and rules," in *Proceedings of The Third CIPS-SIGHAN Joint Conference on Chinese Language Processing*, 2014, pp. 114–119.
[51] 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.

Hongfa Xue is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering, The George Washington University. His research interests are System security and Machine Learning optimization.

Guru Venkataramani (SM '15) received the Ph.D. degree from the Georgia Institute of Technology, Atlanta, in 2009. He has been an Associate Professor of Electrical and Computer Engineering with The George Washington University since 2009. His research area is computer architecture, and his current interests are hardware support for energy/power optimization, debugging, and security. He was a general chair for HPCA'19 and a recipient of the NSF Faculty Early Career Award in 2012.

Tian Lan received the Ph.D. degree from the Department of Electrical Engineering, Princeton University, in 2010. He joined the Department of Electrical and Computer Engineering, The George Washington University, in 2010, where he is currently an Associate Professor. His interests include mobile energy accounting, cloud computing, and cyber security. He received the best paper award from the IEEE Signal Processing Society 2008, the IEEE GLOBECOM 2009, and the IEEE INFOCOM 2012.