Clone-Slicer: Detecting Domain Specific Binary Code Clones through Program Slicing

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

Guru Venkataramani
The George Washington University
Washington, DC, USA
guru@gwu.edu

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

ABSTRACT

Detecting code clones is important for various software engineering development and debugging tasks. In particular, binary code clone detection can have significant uses in the context of legacy applications that are already deployed in several critical domains.

In this paper, we present an novel framework, Clone-Slicer, for identifying domain-specific binary code clones (e.g., pointer-related code) through program slicing. Our approach first eliminates non-domain-related instructions through program slicing, and then applies deep learning-based algorithm to model code samples as numerical vectors for the remaining binary instructions. We then use clustering algorithms to aggregate code clones, and use formal analysis to verify validity of code clones. Our experimental results show the Clone-Slicer can swiftly identify up tp 43.64% code clones and cut the time-to-solution by 32.96% compared to previously proposed code clone detectors.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Security and privacy** → **Software security engineering**;

KEYWORDS

Program slicing; Code Clones; Machine learning; Binary analysis

ACM Reference Format:

Hongfa Xue, Guru Venkataramani, and Tian Lan. 2018. Clone-Slicer: Detecting Domain Specific Binary Code Clones through Program Slicing. In *The 2018 Workshop on Forming an Ecosystem Around Software Transformation (FEAST '18), October 19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3273045.3273047

1 INTRODUCTION

Understanding software and detecting duplicate code fragments is an important task, especially in large code bases [15, 24, 29]. Detecting similar code fragments, usually referred to as *code clones*, can be helpful in discovering vulnerability, refactoring code and removing unnecessary code segments. Prior approaches have been proposed for code clone detection that take advantage of token subsequence matching, text/tree comparison or control flow graph analysis [6, 21, 23]. While a number of existing clone detection algorithms target source code [7, 25, 41], we note that legacy applications exist in several real-world domains and have been in deployment for a number of years in production systems including airspace, military and banking (where only binary executables are available). Also, binary code clone detection is more difficult compared to source code-level detectors that leverage rich structural information such as syntax trees and variable names made available through the source lines of program code.

Prior work on binary code clones have adapted source code-based techniques and are usually oblivious to the specific domain of applications on which clone detection is useful. For instance, Sæbjørnsen et al [33] have proposed a code clone detection algorithm based on characteristic vectors and normalize the assembly-level instructions to detect more clones. Xu et al. [43] propose a graph embedding approach using deep neural network to detect vulnerable binaries that are compiled for different computer architectures or platforms. While these prior work outline methods to detect code clones, it is *more* important to *tailor* code clone detection based on *specific application areas* to increase usefulness of detecting them.

To improve the application of detecting code clones, we introduce domain-specific code clone detection, which can be used to detect code clones for certain types of applications. This approach takes advantage of the knowledge within a specific domain and tailors code clone detection approach based on that domain. For instance, since pointers and pointer-related operations widely exist in real-world applications and are often behind security bugs [9, 11, 34], detecting code clones related to pointers are of great significance for application security. Thus, detecting pointer-related code clones is one such domain-specific application that can significantly improve the scalability for pointer-specific application analysis. Similarly, tracking code that depends on external inputs is another example of domain-specific code.

In this paper, we propose Clone-Slicer, a novel framework for domain specific code clone detection in binaries. In particular, we select pointer analysis (that determines pointer safety) to present our methodology and demonstrate the soundness of our approach in this work. We first deploy a lightweight pointer tainting method in binary to find pointer-related instructions that can potentially change the array boundary conditions. Then we leverage forward program slicing and binary rewriting to remove pointer-irrelevant instructions in order to detect pointer-related code clones. By doing so, we are able to improve the number of code clones detected and their time-to-solution in determining applicability on a specific domain (removing unnecessary checking code surrounding pointers that are already deemed safe). We note that this enables rapid security analysis by ignoring binary instructions that are irrelevant to pointer-related code.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

FEAST '18, October 19, 2018, Toronto, ON, Canada

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-5997-9/18/10...$15.00
https://doi.org/10.1145/3273045.3273047

The contributions of our paper are summarized as follows:

(1) We propose Clone-Slicer, a domain-specific binary code clone detection framework. Given a domain of interest, Clone-Slicer automatically identifies domain-related binary instructions through tainting frameworks, and then performs code clone detection to enable rapid security analysis.

(2) Clone-Slicer leverages program slicing to remove domain-irrelevant instructions to find code clones of interest within the application domain. Clone-Slicer deploys formal analysis to perform a closed-loop operation and introduces a clone verification mechanism to formally verify if identified clone samples are indeed clones within the domain context.

(3) We implement a prototype of Clone-Slicer and evaluate pointer analysis domain using real-world applications from SPEC2006 benchmarks suite [1]. Our results show Clone-Slicer can swiftly identify up tp 43.64% code clones and cut the time-to-solution for (the time spent to formally verify the redundancy of array bound checks) by 32.96% compared to prior work [45].

The rest of this paper is structured as follows: In Section 2, we survey related work. We illustrate the overview of Clone-Slicer and how we design and implement our system, respectively. We evaluate Clone-Slicer and show our experimental results in Section 4. Section 5 discusses our conclusions and future work.

2 RELATED WORK

Code clone detection. Code clone detection techniques can generally be classified into several categories. String matching-based techniques [5, 6, 13] apply lightweight program transformations and utilize code similarity measurement through comparing text sequences of text. Such text-based techniques are limited in scalability for large code bases and only find exact match code clone pairs. Second, tree- or token-based clone detection [7, 26, 41] are performed by parsing program into tokens or generate abstract syntax tree (AST) representation of the source program. Consequently, tree- or token-based approaches usually more robust against code-specific changes. Some well-known tools in this category include CC-Finder [23], DECKARD [21] and CP-Miner [29]. Learning based approaches have also been developed for code similarity detection. White et al. [42] proposed a deep neural network (DNN) based code clone detection in source code. Komondoor et al. [25] also make the use of program slicing and dependence analysis to find non-contiguous code clones. But such approaches typically find isomorphic subgraphs from program dependency graph in order to identify code clones, for which computing such graphs is typically more expensive. Also, the approaches mentioned above are still demonstrated on the source code-level and not on binaries. Gemini [43] use DNN to detect cross-platform code clones in binaries. But it is limited in scope to detect clones within a single function complied in different platforms.

Statistical method and Formal analysis. In this paper, we make use of both machine learning and formal analysis for code clone detection and verification. Prior work have studied bug/vulnerabilities using learning based approaches [20, 32]. StatSym [46] and SARRE [28] propose frameworks combining statistical and formal analysis for vulnerable path discovery. SIMBER [44] proposes a statistical inference framework to eliminate redundant bound checks and improve the performance of applications without sacrificing security. However, SIMBER is limited in scalability and does not use machine learning algorithms. Similarly, Clone-Hunter [45] also takes advantage of binary code clone detection for accelerated bound check removal. However, Clone-Hunter uses normalization on instruction operand for vector embedding, which may omit relevant information and does not use domain-specific knowledge (e.g., pointers) for improved clone detection. In this paper, we use deep learning based language models for vector embedding instead.

Deep Learning and Language Modeling. The state-of-the-art Deep Learning algorithms have been used as new approaches for language modeling [3, 22]. Traditionally, natural language processing (NLP) in particular has utilized deep learning to do software engineering tasks such as text/code suggestions, text classification and so on [3, 14, 19]. For instance, recurrent neural network (RNN) is known as a capable approach for modeling sequential information [17, 36]. Recently, such techniques has been applied on modeling program source code fragments. White et al. [42] propose a deep learning-based detection approach for source code clone detection using RNN. It develops an automated framework to extract source code features at both lexical and syntax levels. To the best of our knowledge, we are the first to demonstrate improved code clone detection in a scalable way using deep learning and clustering algorithms, while making sure that clones are verified through formal analysis in the back end.

3 SYSTEM DESIGN AND IMPLEMENTATION

In this section, we present the overview and details of our system design along with its modules, and show how our system is implemented. The kernel of Clone-Slicer is shown in Figure 1.

For a given application binary, Clone-Slicer first employs static binary program slicing and binary rewriting to remove pointer irrelevant instructions. We disassemble binary executables and work with the resulting assembly code (Section 3.1). To detect code clones in binaries, we leverage deep learning-based approach to generate feature vectors for each instruction sequence and embed them into vector space (Section 3.2). After we obtain feature vectors, we deploy clustering algorithm to form clusters and find code clone pairs. Note that we also use different code similarity thresholds to further increase the number of detected code clones (more details in Section 3.3). Since we adopt sliding window-based method to generate code regions, we perform quick post-processing to consolidate overlapping code clones.

We use binary symbolic execution to verify whether the code clone samples are safe in terms of array bound checks. We deploy a selective sampling method to further verify the validity of clone detection by selecting a random subset of samples within the cluster center and boundary regions, and perform binary symbolic execution on these samples. Section 3.4 describes our implementation in more detail.

3.1 Domain Specific Program Slicing

In pointer analysis domain, we aim to analyze each pointer in the program to ensure there is no issue like memory violation. Thus, only some certain types of instructions are related to the target pointer for further consideration, which can affect the base, offset or bound information of this pointer. In this paper, we use pointer tainting analysis to find such pointer-related instructions at a function-level granularity. Then, we deploy forward program slicing and binary rewriting to remove pointer irrelevant instructions.

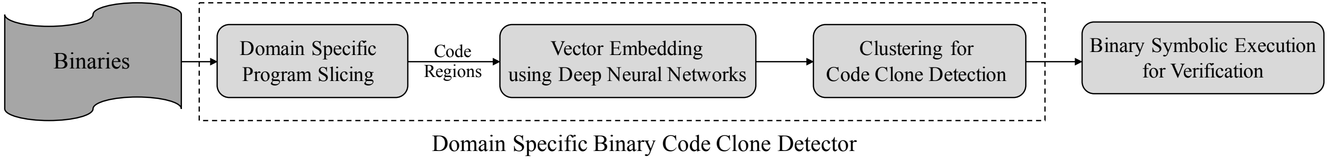

Figure 1: The Kernel of Clone-Slicer

To address this problem, Clone-Slicer first performs tainting analysis of the binary code and deploy program slicing in two steps:

(1) **Lightweight Pointer Tainting.** To select pointer related instructions, we utilize a lightweight pointer tainting mechanism. Typically, there are two types of instructions need to be tainted: Memory load operations moving data from memory to register; Store operations moving data from register to memory.We implemented the pointer tainting based on previous work [10, 39, 40]. Whenever a program performs memory operations using its data from registers and memory, such instructions need to be tainted through propagation. In particular, for each load instruction, the tainting is propagated from memory to register along the load path. Similarly, for each store instruction, the tainting is propagated from writing to the memory along the store path. Whenever two pointers are subtracted (e.g., offset computation), the resulting location is un-tainted. However, addition of two pointers still results in a pointer.

(2) **Program Slicing.** After we obtain all the target pointers and their corresponding pointer-related instructions, we use forward program slicing and binary rewriting to remove pointer-irrelevant instructions. To build a forward slice, we utilize control flow graph (CFG) and data dependency graph (DDG) to understand the dependency among all the tainted instructions. Forward slicing is then constructed starting with tainted targets in the program, and all of the data flows in this slice end at the target after traversing the entire CFG. We then are able to select all pointer-related instructions. For those instructions are pointer irrelevant, we simply rewrite them as *nop* using binary rewriting tools.

To remove pointer irrelevant instructions in binary executables, we deployed a Static Binary Rewriting tool Dyninst [37]. We instrumented a binary analysis framework angr [35] and develop a python script to construct CFG and DDG in binaries.

3.2 Vector Embedding using Deep Neural Networks

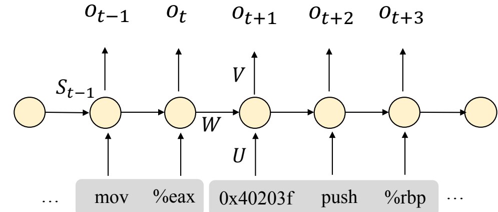

Figure 2: An illustration of RNN. The input of each node is a one-hot vector representing the current term in the disassembly code corpus, and output is a probability distribution predicting the next term. U, V, W are the parameters in the network, and s_t is the hidden layer state vector.

We adopt a sliding window method to select different code regions for code clone analysis. The approach is implemented with two parameters: window size and stride. Window size defines the maximum length of code regions for consideration, while stride denotes the smallest increment of starting instruction address for subsequent sliding windows. Since we rewrite non-pointer related instructions as *nop*. We skip such *nop* instructions while we generate code regions and only count pointer related sliced instructions in the code regions.

Next, we leverage Deep Neural Network (DNN) to propose a solution to enable automated vector embedding. First, to obtain vector embedding for a given code region (that consists of an instruction sequence), we use Recursive Neural Network (RNN) to map each term in the binary instructions (e.g., opcodes and operands) to a vector embedding at lexical level, resulting in a signature vector for the code region.

Embedding binary code at lexical level. Consider a disassembly code corpus from a target program, with m distinct terms (e.g., different opcodes and operands) across the whole corpus. We use a RNN with n hidden nodes to convert each term in the code corpus into an embedding vector $U \in \mathbb{R}^{n \times m}$. RNN is known as an effective approach for modeling sequential information, such as sentences in texts or program code. Figure 2 presents the training process of our RNN model for binary code. The input $x_t \in \mathbb{R}^{m+n}$ at time step t is a one-hot vector representation [38] corresponding to the current term, e.g., *'exa'*. The hidden layer state vector, $s_t \in \mathbb{R}^n$, stores the current state of the network at step t and captures the information that has already been calculated. Specifically, it can be obtained using the previous hidden state s_{t-1} at time step $t-1$ and the current input x_t at time step t:

$$s_t = f(Ux_t + Ws_{t-1}) \quad (1)$$

Function f is a nonlinear function, e.g., $tanh.U \in \mathbb{R}^{n \times m}$ and $W \in \mathbb{R}^{n \times n}$ are the shared parameters in all time steps.

The output, $O_t \in \mathbb{R}^m$, is a vector of probabilities predicting the distribution of the next term in the code corpus [18]. It is calculated based on current state vector along with another shared parameter $V \in \mathbb{R}^{m \times n}$, i.e., :

$$O_t = softmax(Vs_t) \tag{2}$$

The parameters $\{U, V, W\}$ are trained using back propagation through time (BPTT) method in our RNN network [8]. Once RNN training is complete, each term in the code corpus will have an unique embedding U from Equation (1), which comprises its semantic representation cross the corpus [4]. We compute such embeddings U to represent the terms of binary instructions at lexical level.

Generating signature at syntax level. We use Autoencoder to combine embeddings $U \in \mathbb{R}^{nm}$ of the terms from multiple instructions and to obtain a signature vector for a given code region. Autoencoder is widely used to generate vector space representations for a pairwise composed terms with two phases: encode phase and decode phase. It is a simple neural network with one input layer, one hidden layer and one output layer. As shown in Figure 3, we apply Autoencoder recursively to a sequence of terms, which is known as the Recursive Autoencoder (RAE). Let $x_1, x_2 \in \mathbb{R}^{nm}$ be the vector embeddings of two different terms, computed using RNN. During encode phase, the composed vector embeddings $Z(x_1, x_2)$ is calculated by:

$$Z(x_1, x_2) = f(W_1[x_1; x_2] + b_1), \tag{3}$$

where $[x_1; x_2] \in \mathbb{R}^{2nm}$ is the concatenation of x_1 and x_2, $W_1 \in \mathbb{R}^{nm \times 2nm}$ is the parameter matrix in encode phase, and $b \in \mathbb{R}^{nm}$ is the offset. Similar to RNN, f again is a nonlinear function, e.g., $tanh$. In decode phase, we need to assess if $Z(x_1, x_2)$ is well learned by the network to represent the composed terms. Thus, we reconstruct the the term embeddings by:

$$O[x_1; x_2] = g(W_2[x_1; x_2] + b_2), \tag{4}$$

where $O[x_1; x_2]$ is the reconstructed term embeddings , $W_2 \in \mathbb{R}^{nm \times 2nm}$ is the parameter matrix for decode phase, and $b_2 \in \mathbb{R}^{nm \times 1}$ is the offset for decode phase and the function g is another nonlinear function. For training purpose, the reconstruction error is used to measure how well we learned term vector embeddings. Let $\theta = \{W_1; W_2; b_1; b_2\}$. We use the Euclidean distance between the inputs and reconstructed inputs to measure reconstruction error, i.e.,

$$E([x_1; x_2]; \theta) = ||[x_1; x_2] - O[x_1; x_2]||_2^2 \tag{5}$$

For a given code region with multiple terms and instructions, we adopt a greedy method [42] to train our RAE and recursively combine pairwise vector embeddings. The greedy method uses a hierarchical approach – it first combines vector embeddings of adjacent terms in each instructions, and then combines the results from a sequence of instructions in an execution path. Figure 3 shows an example of how to combine the vector embeddings to generate a signature vector. It shows a (binary) execution path with a sequence of 8 instructions. The greedy method is illustrated as a binary tree. Node 1 gives the vector embedding for the first instruction $Inst_1 = (push\ \%rbp)$ encoded from terms $[push; \%rbp]$. Then, we continue to process the remaining instructions, e.g., Nodes 2 and 3, until we derive the final vector embedding (i.e., the signature vector) for the instruction sequences of the given execution path.

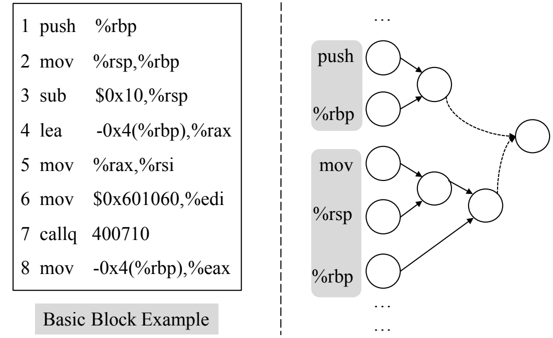

Figure 3: RAE combines embeddings from different terms and instructions through a Greedy method.

We used IDA Pro [2] for disassembly and implemented RNN and RAE in python based on the framework proposed in [27]. For RNN, we develop a python script to tokenize the disassembly code and use the RNNLM Toolkit [30] to train RNN for each program, with the hidden layer size equal to 500.

3.3 Clustering for Code Clone Detection

3.3.1 Definitions. We first formally give the definition of code similarity used in our code clone detection module.

Definition 3.1. **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 following:

$$S(T_1, T_2) = \frac{2S}{2S + L + R} \tag{6}$$

where S is the number of shared nodes in T_1 and T_2, L and R are the different nodes in terms of the node types and number of nodes in T_1 and T_2 respectively.

3.3.2 Clone Detection. Given a group of feature vectors, we utilize Locality Sensitive Hashing (LSH) [12] 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 [16, 21]. 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:

$$E([V_i; V_j]) = ||V_i - V_j||_2^2 \tag{7}$$

$$H([V_i; V_j]) = ||V_i - V_j||_1 \tag{8}$$

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:

$$E([V_i; V_j]) \geq \sqrt{H([V_i; V_j])} \approx \sqrt{L + R} \tag{9}$$

Based on the definition from Equation 3.1, 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:

$$T = \sqrt{2(1-S) \times min(S(C_i), S(C_j))} \quad (10)$$

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 3.1. Thus, code fragments C_i and C_j will be clustered together as code clones under a given code similarity S if $E([V_i; V_j]) \leq T$.

3.3.3 Post-Processing. As described in previous section, we deploy a sliding window approach to generate code fragments for code clone detection. We note that this method can potentially create duplicated or overlapping code clones. To address this problem, we further eliminate such code clones and only preserve the largest code clones by computing the union of overlapping code clones. Assuming a code clone sample is denoted as (c_1, c_2) , where c_1 is the starting instruction address of the code and c_2 is the ending instruction address in the code fragment. Give two code clone samples (c_1, c_2) and (c_1', c_2'), we only keep clone sample (c_1, c_2) iff $c_1' \geq c_1$ and $c_2' \leq c_2$. On the other hand, we do not consolidate two code clone samples if $(c_1, c_2) \cap (c_1', c_2') \neq (c_1, c_2) or (c_1', c_2')$. This post-processing procedure is performed until all consolidated code clones are non-overlapping.

We implemented our clustering system with python and provide as a user-friendly interface in Linux command line, which can provide the options of code similarity S for users.

3.4 Binary Symbolic Execution for Verification

Clone-Slicer makes use of clustering algorithms to identify binary code clones. In prior work Clone-Hunter [45] , it uses binary code clone detection to assist removal of redundant array bound checks. Clone-Slicer can be further applied to the same task to remove redundant bound checks. Similarly, we utilize binary symbolic execution to formally verify if the code clone samples are memory safe in the same cluster.

There are two major steps for this verification process in Clone-Slicer:

(1) **Selection of samples for analysis:** First, we pick a random code clone sample from each cluster center as *seed code sample*. We determine the pointer dereference is safe, and that no memory violation can exist. We deploy partial binary symbolic execution to execute the *seed code sample*, which we perform symbolic execution starting from beginning to end of the seed code sample based on its instruction addresses. We check whether the code samples contain memory violation (e.g. buffer overflow) based on the output from symbolic execution.
Note that this identification process can be further applied to the task like redundant bound checks removal. If the pointers in *seed code sample* turn out to be safe, then array bound checks may be safely removed. To the contrary, the bound checks cannot be removed if the output from symbolic execution says that there are memory violation in the corresponding code snippet. We further conduct a case study applying the kernel of Clone-Slicer to redundant bound check removal to show the applications of Clone-Slicer(Section 4.4)

(2) **Verification of memory safety:** Since machine learning based clustering algorithm cannot offer any guarantees in terms of ensuring memory safety from all detected code clones. It is possible the code clone samples have different memory safety conditions in the same cluster. To address such issue, Clone-Slicer further executes a verification process. We select a random set of code clone samples from the cluster boundary within the same cluster and perform the same partial binary symbolic execution to check whether the memory safety conditions on these code clones are indeed similar. If the random code clones samples also turn out to be safe just as the *seed code sample* does, then we assume all the code clone samples are safe in the corresponding cluster.

We instrumented a binary analysis framework angr [35] for our verification module. We take advantage of the binary symbolic executor in angr to perform partial symbolic execution, which is beginning with the starting address and execute instructions within the specific code region to the end.

4 EVALUATION

In this section, we provide an overview of our experimental setup. We later present our evaluation results in terms of the effectiveness of code clone detection using our approach and the overhead of binary symbolic execution comparing to prior work, Clone-Hunter [45].

4.1 Experiment Setup

We performed empirical experiments on Clone-Slicer. 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. We selected 4 different real-world applications: hmmer, sphinx3, bzip2 and lbm from SPEC2006 benchmark suite [1].

4.2 Code Clone Detection

We measured the number of code clones that are detected from Clone-Slicer using domain-specific knowledge (pointer safety, in our case). We conduct experiments in terms of the following: code clones quantity and the effect of relaxing the code similarity metric. We use the binary code clone detection algorithm proposed in Clone-Hunter as our baseline. For a fair comparison, we choose the same configuration to generate code regions with maximum sliding window size equals to 100 instructions (minimum window size = 2 instructions) and stride value of 4.

Table 2 shows the experiment results for each benchmark, the number of code clone detected using Clone-Hunter and Clone-Slicer, with code similarity thresholds equal to 1.00 and 0.90. First, we are able to increase the number of code clone detection while we relax the code similarity. Second, as we can say, Clone-Slicer is able to detect more code clones than Clone-Hunter among all the benchmarks, with the highest up to 43.64% improvement than Clone-Hunter.

4.3 Overhead of Binary Symbolic Execution

We evaluated the overhead of binary symbolic executors to check for pointer memory safety using Clone-Slicer, and compared the execution time with Pure Symbolic Execution on function-level (performing partial symbolic execution on each function as function-level overhead) and Clone-Hunter. Similarly, our baseline is the binary analysis framework angr [35].

For a fair comparison, we set up the same threshold for number of code samples used for verification as mentioned in Clone-Hunter,

Table 1: Comparison of execution time spent in Pure Symbolic Execution, Clone-Hunter and Clone-Slicer

Benchmark	Program Size (Byte)	Pure Symbolic Execution Time Function-Level (sec)	Clone-Hunter assisted Symbolic Execution time (sec)	Clone-Slicer assisted Symbolic Execution time (sec)	%Improvement of time-to-solution
bzip2	305K	383.4	154.0	103.2	32.96%
lbm	55K	1584.4	387.9	308.5	20.45%
hmmer	974K	6733.3	957.4	710.7	25.76%
sphinx3	1.3M	14010.0	6144.3	5202.2	15.33%

Table 2: Comparison of number of code clones detected by Clone-Slicer and Clone-Hunter

Benchmark	#Code Clones		%Improvement
	Clone-Hunter	Clone-Slicer	
bzip2	27	37	37.04%
lbm	10	14	40.00%
hmmer	261	352	34.44%
sphinx3	1,488	1,815	21.98%

Code Similarity Threshold, $S = 1.00$

Benchmark	#Code Clones		%Improvement
	Clone-Hunter	Clone-Slicer	
bzip2	55	79	43.64%
lbm	32	40	25.00%
hmmer	587	769	31.10%
sphinx3	1,988	2,417	21.58%

Code Similarity Threshold, $S = 0.90$

with a lower bound as 2 code clone samples (since the smallest cluster only contains two code clone samples) and 30% sampling rate for larger cluster as upper bound to randomly select code clone samples described in Section 3.4. Table 1 presents the runtime overhead due to pure symbolic execution on function-level, Clone-Hunter and Clone-Slicer. We observe that Clone-Slicer is able to improve the time-to-solution (the time spent to verify pointer memory safety) comparing to Clone-Hunter among all the testing benchmarks, with the highest up to 32.96% improvement of time-to-solution in bzip2.

4.4 Case Study: Removing Redundant Array Bound Checks

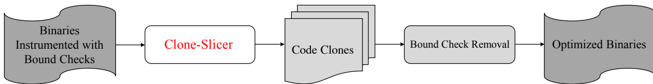

Figure 4: Application of Clone-Slicer kernel to remove redundant bound checks

As mentioned in previous sections, Clone-Slicer proposes a memory safety verification mechanism after detecting code clones which can be further used in different engineering tasks. Here, we applied the kernel of Clone-Slicer for redundant bound checks removal task. We selected two representative benchmarks: bzip2 and sphinx3 to present the results. Figure 4 shows the process of redundant bound checks removal. We use Clone-Slicer on the top of binaries instrumented with bound checks and identify code clones with code similarity equaling to 0.90. Clone-Slicer is able to automatically verify whether bound checks are redundant in the code clones (if binary symbolic execution raises no memory violation). Afterwards, we deploy a static binary rewriter Dyninst [37] to remove bound checks in binaries.

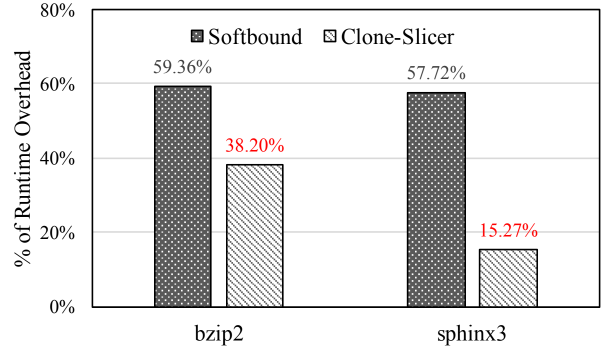

Figure 5: Runtime overhead of softbound-instrumented applications and Clone-Slicer. The baseline is non-instrumented applications.

To evaluate the performance of Clone-Slicer, we employ a runtime bound checker tool: Softbound [31] to insert bound checks in the benchmarks. Figure 5 shows the comparison of Softbound's runtime execution overhead before and after using Clone-Slicer. Our results show that Clone-Sliceris able to significantly reduce the runtime overheads caused by redundant array bound checks in both bzip2 and sphinx3. Clone-Slicer achieves the highest overhead reduction up to 42.25% in sphinx3.

5 CONCLUSION AND FUTURE WORK

In this paper, we presented a novel framework, Clone-Slicer, a domain-specific code clone detector for binary executables, that integrates program slicing and a deep learning based binary code clone modeling framework to improve the number of code clone detected. In particular, we chose pointer analysis for memory safety as our example domain to demonstrate the usefulness of our approach. We evaluated our approach using real-world applications from SPEC 2006 benchmark suite. Our results show Clone-Slicer is able to detect up to 43.64% code clones compared to prior work and further cut the time-to-solution (the time spent to verify memory bound safety) for Clone-Slicer by 32.96% compared to Clone-Hunter.

As future work, we plan to apply Clone-Slicer to different domains and tasks, such as vulnerable program path discovery, and further improve the capability for code clone detection through

advanced clustering algorithms. We will also study the cost-benefit tradeoffs of using such advanced algorithms.

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] 2006. SPEC CPU 2006. https://www.spec.org/cpu2006/.
[2] 2016. IDA Pro disassembler. https://www.hex-rays.com/products/ida/.
[3] Sheeva Afshan, Phil McMinn, and Mark Stevenson. 2013. Evolving readable string test inputs using a natural language model to reduce human oracle cost. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. IEEE, 352–361.
[4] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 38–49.
[5] Brenda S Baker. 1995. On finding duplication and near-duplication in large software systems. In *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*. IEEE, 86–95.
[6] Brenda S Baker. 1997. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. Comput.* 26, 5 (1997), 1343–1362.
[7] Ira D Baxter, Christopher Pidgeon, and Michael Mehlich. 2004. DMS/spl reg: program transformations for practical scalable software evolution. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*. IEEE, 625–634.
[8] Christopher M Bishop. 2006. Machine learning and pattern recognition. *Information Science and Statistics. Springer, Heidelberg* (2006).
[9] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. 2012. 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, 133–143.
[10] Shuo Chen, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar K Iyer. 2005. Defeating memory corruption attacks via pointer taintedness detection. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*. IEEE, 378–387.
[11] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. 2015. 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, 952–963.
[12] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*. ACM, 253–262.
[13] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. 1999. A language independent approach for detecting duplicated code. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*. IEEE, 109–118.
[14] Christine Franks, Zhaopeng Tu, Premkumar Devanbu, and Vincent Hellendoorn. 2015. Cacheca: A cache language model based code suggestion tool. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 705–708.
[15] Mark Gabel and Zhendong Su. 2010. A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 147–156.
[16] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. 1999. Similarity search in high dimensions via hashing. In *Vldb*, Vol. 99. 518–529.
[17] Christoph Goller and Andreas Kuchler. 1996. Learning task-dependent distributed representations by backpropagation through structure. In *Neural Networks, 1996., IEEE International Conference on*, Vol. 1. IEEE, 347–352.
[18] Vincent J Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 763–773.
[19] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 837–847.
[20] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2017. Binary code clone detection across architectures and compiling configurations. In *Proceedings of the 25th International Conference on Program Comprehension*. IEEE Press, 88–98.
[21] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 96–105.
[22] Dan Jurafsky and James H Martin. 2009. Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition. , 1024 pages.
[23] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670.
[24] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. 2005. An empirical study of code clone genealogies. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 187–196.
[25] Raghavan Komondoor and Susan Horwitz. 2001. Using slicing to identify duplication in source code. In *International Static Analysis Symposium*. Springer, 40–56.
[26] Kostas A Kontogiannis, Renator DeMori, Ettore Merlo, Michael Galler, and Morris Bernstein. 1996. Pattern matching for clone and concept detection. *Automated Software Engineering* 3, 1-2 (1996), 77–108.
[27] Peng Li, Yang Liu, and Maosong Sun. 2013. Recursive autoencoders for ITG-based translation. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. 567–577.
[28] Yongbo Li, Fan Yao, Tian Lan, and Guru Venkataramani. 2016. Sarre: semantics-aware rule recommendation and enforcement for event paths on android. *IEEE Transactions on Information Forensics and Security* 11, 12 (2016), 2748–2762.
[29] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2006. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering* 32, 3 (2006), 176–192.
[30] Tomas Mikolov, Stefan Kombrink, Anoop Deoras, Lukar Burget, and Jan Cernocky. 2011. Rnnlm-recurrent neural network language modeling toolkit. In *Proc. of the 2011 ASRU Workshop*. 196–201.
[31] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. *ACM Sigplan Notices* 44, 6 (2009), 245–258.
[32] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-architecture bug search in binary executables. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 709–724.
[33] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. 2009. Detecting code clones in binary executables. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 117–128.
[34] Fermin J Serna. 2012. The info leak era on software exploitation. *Black Hat USA* (2012).
[35] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 138–157.
[36] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Ng, and Christopher Potts. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*. 1631–1642.
[37] Open Source. 2016. Dyninst: An application program interface (api) for runtime code generation. *Online, http://www. dyninst. org* (2016).
[38] Joseph Turian, Lev Ratinov, and Yoshua Bengio. 2010. Word representations: a simple and general method for semi-supervised learning. In *Proceedings of the 48th annual meeting of the association for computational linguistics*. Association for Computational Linguistics, 384–394.
[39] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. 2008. Flexitaint: A programmable accelerator for dynamic taint propagation. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*. IEEE, 173–184.
[40] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. 2009. MemTracker: An accelerator for memory debugging and monitoring. *ACM Transactions on Architecture and Code Optimization (TACO)* 6, 2 (2009), 5.
[41] Vera Wahler, Dietmar Seipel, J Wolff, and Gregor Fischer. 2004. Clone detection in source code by frequent itemset techniques. In *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*. IEEE, 128–135.
[42] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 87–98.
[43] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. 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, 363–376.
[44] Hongfa Xue, Yurong Chen, Fan Yao, Yongbo Li, Tian Lan, and Guru Venkataramani. 2017. Simber: Eliminating redundant memory bound checks via statistical inference. In *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 413–426.
[45] Hongfa Xue, Guru Venkataramani, and Tian Lan. 2018. 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, 11–19.
[46] Fan Yao, Yongbo Li, Yurong Chen, Hongfa Xue, Tian Lan, and Guru Venkataramani. 2017. Statsym: vulnerable path discovery through statistics-guided symbolic execution. In *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on*. IEEE, 109–120.