Learn2Reason: Joint Statistical and Formal Learning Approach to improve the Robustness and Time-to-solution for Software Security

by Hongfa Xue

A Dissertation submitted to

The Faculty of
The School of Engineering and Applied Science
of the George Washington University
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

January 10, 2020

Dissertation directed by

Guru Prasadh Venkataramani
Associate Professor of Engineering and Applied Science

The School of Engineering and Applied Science of The George Washington University certifies that Hongfa Xue has passed the Final Examination for the degree of Doctor of Philosophy as of October 16, 2019. This is the final and approved form of the dissertation.

Learn2Reason: Joint Statistical and Formal learning Approach to improve the Robustness and Time-to-solution for Software Security

Hongfa Xue

Dissertation Research Committee:

Guru Prasadh Venkataramani, Associate Professor of Engineering and Applied Science, Dissertation Director

Tian Lan, Associate Professor of Engineering and Applied Science, Committee Member

Howie Huang, Professor of Engineering and Applied Science, Committee Member

Payman Dehghanian, Assistant Professor of Engineering and Applied Science, Committee Member

Henrique Aveiro, Section Head, Procter & Gamble, Committee Member

Dedicated to my beloved mother and my D.C. families

Acknowledgements

First, I would like to thank my advisor, Professor Guru Prasadh Venkataramani for his tremendous guidance, patience and support throughout my Ph.D. program. His expertise and knowledge have always inspired me to the new ideas for my research papers and works. As an advisor, he is always believing in me even when I was going through the darkest time in my life. Professor Venkataramani has become an awesome mentor and friend. He gave me so much valuable advice that helps me keep positive and never give up. I feel much blessed to have him as my advisor.

I would like to thank my dissertation committee members, Prof. Tian Lan, Prof. Howie Huang, Prof. Payman Dehghanian and Dr. Herique Aveiro, who have truly guided me for my successes inside and outsides my academic career. I truly feel fortunate to have worked with many of them. Especially, I have been working with Professor Lan since the beginning of my Ph.D. program. He has provided me extensive important insights for my research and personal guidance for my personal life.

I also would like to thank my friends and colleagues in GWU, Fan Yao, Yongbo Li, Jingxin Wu, Hongyu Fang, Yurong Chen, and many others, who have filled my Ph.D. journey with lots of fun and joy.

Finally, I would like to thank my parents, Baixiang Ru and Yanqun Xue for their relentless love and unconditional support. I would not get so far without them and I am deeply grateful to them.

This dissertation is based upon work supported by the US Office of Naval Research (ONR) under Awards N00014-15-1-2210 and N00014-17-1-2786.

Abstract of Dissertation

Learn2Reason: Joint Statistical and Formal learning Approach to improve the Robustness and Time-to-solution for Software Security

Thesis Statement: Past research have proposed techniques for finding vulnerabilities in software applications using either statistical analysis (fast but may be inaccurate) or formal analysis (accurate but may be slow). In this dissertation, we propose Learn2Reason, a novel joint learning approach that harnesses the advantages of both statistical analysis and formal techniques to improve the robustness and time-to-solution for software security.

With the rapid rise in software sizes and complexity, analyzing and fixing bugs in large scale applications is becoming increasingly critical, securing such application has become very challenging due to the growing software complexity. Traditionally, there are two lines of code analysis techniques that have some fundamental limitations: *Pure statistical methods* and *Pure formal methods*. Using solely either lacks accuracy or require exhaustive analysis along all paths in the application code.

In this dissertation, we first design a joint learning framework using both techniques to protect unsafe memory accesses in programs. As today, such memory violation issues have been among the leading causes of software vulnerability. Memory safety checkers, such as Softbound, enforce memory spatial safety by checking if accesses to array elements are within the corresponding array bounds. However, such checks often result in high execution time overhead due to the cost of executing the instructions associated with the bound checks. To mitigate this problem, techniques to eliminate redundant bound checks are needed. We propose two novel frameworks, SIMBER and Clone-Hunter to eliminate redundant memory bound checks in source code and binaries respectively. In contrast to the existing techniques that primarily rely on static code analysis, our solution leverages learning-based techniques to identify redundant bound checks.

Additionally, understanding software and detecting duplicate code fragments is an important task, especially in large code bases. Detecting similar code fragments, usually referred to as *code clones*, can be helpful in discovering vulnerabilities, refactoring code and removing unnecessary code segments. In particular, binary code clone detection can have significant uses in the context of legacy applications that are already deployed in several critical domains. We present learning based frameworks for Domain-Specific Code Clone Detection. 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 the validity of code clones.

To further illustrate the benefit of our joint learning approach, we leverage machine learning-based binary code analysis frameworks, combined with dynamic execution and trace analysis to create customized, self-contained programs, in order to minimize the potential attack surface. It automatically identifies program features (i.e., independent, well-contained operations, utilities, or capabilities) relating to application binaries and their communication functions, tailors and eliminates the features to create customized program binaries in accordance with user needs, in a fully unsupervised fashion.

This dissertation aims to harnesses the advantages of both statistical analysis and formal techniques to perform rigorous code analysis in both source code and binary executables while maintaining scalability and swiftness. The main contributions of this dissertation are to improve the security issues of code analysis by integrating statistical analysis and formal methods, thereby reducing the time-to-solution.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Program Code Analysis for Software Security

With the rapid growth of software systems, securing such applications has become very challenging due to the growing software complexity. Prior studies have shown that there are about 5 to 20 bugs per 1,000 lines of software. Exploitation of such bugs and program vulnerabilities through source code or binary analysis has been a major threat to computer security and user data safety.

Traditionally, there are two lines of code analysis techniques that have some fundamental limitations. 1. *Pure statistical methods* rely on probabilistic inference and often fail to guarantee complete accuracy. Any conclusions derived from sampling the runtime program states can offer only limited visibility, and are prone to false alarms. As a result, considerable human effort is still required to verify the results from statistical analysis. 2. *Pure formal methods* require exhaustive analysis along all paths in the application code, which can be prohibitively expensive in terms of time and resources. As such, strict symbolic execution methods can be less effective in analyzing software at-scale.

1.2 A Joint Learning Approach

In this dissertation, we propose a joint learning approach, an integration of statistical and formal methods without the unification of the underlying knowledge representation. We envision that the decision-making models keeps the cognitive (statistical) and deliberative (formal) levels of reasoning as two separate but interacting modules, with their separate but partially correlated knowledge representations. We hope that the novel approach for synergistically integrating statistical and formal knowledge, in this way enables cross-triggering (prompting), cross-checking and hence allowing both knowledge bases to grow and adapt to the world in synchrony. This property is essential toward achieving up-to-date, comprehensive, timely and accurate planning/decision making. This dissertation harnesses the advantages of both statistical analysis and formal techniques to perform rigorous code analysis in both source code and binary executables while maintaining scalability and swiftness.

The need for memory protection. Many software bugs and vulnerabilities in applications (that are especially written using C/C++) occur due to unsafe pointer usage and out-of-bound array accesses. Security exploits, that take advantage of buffer overflows or illegal memory reads/writes, have been a major concern over the past decade.

To protect software from spatial memory/array bound violations, Bound Checker tools such as Softbound [80, 84, 25, 21, 23] have been developed that maintains metadata such as array boundaries along with rules for metadata propagation when loading or storing pointer values. By doing so, Bound Checkers make sure that pointer accesses do not violate boundaries through runtime checks. While such a tool offers protection from spatial safety violations in programs, we should also note that they often incur high performance overheads due to the following reasons. a) Array bound checking incurs extra instructions in the form of memory loads and stores for pointer metadata and the propagation of metadata between pointers during assignments. b) In pointer-intensive programs, such additional memory accesses can introduce memory bandwidth bottleneck, and further degrade system performance. To mitigate runtime overheads, static techniques to remove redundant checks have been proposed., e.g., ABCD [18] builds and solves systems of linear inequalities among bound and index variables, and WPBound [92] statically computes the potential range of target pointer values inside loops to avoid Softbound-related checks. As the relationship among pointer-affecting variables (i.e., variables, whose values can influence pointers) and array bounds become more complex, static analysis is less effective and usually cannot remove a high percentage of redundant array bound checks.

In order to reduce such high performance overheads, *redundant bound check elimination* approaches have been developed [19, 107, 124, 120, 22, 121]. By eliminating unnecessary bound checks, their corresponding performance overheads can be avoided. However, note that such *redundant array bound check elimination methods still need to analyze every single pointer deference to compute the constraints involv-*

ing pointer-related variables, and verify whether bound checks are redundant and be removed effectively from that location. In the case of applications involving billions of pointer dereferences, the task of verifying the redundancy of bound checks can still be prohibitively expensive or impossible in practice.

The usefulness of detecting code clones. Understanding software and detecting duplicate code fragments is an important task, especially in large code bases [43, 59, 71]. Detecting similar code fragments, usually referred to as *code clones*, can be helpful in discovering vulnerabilities, 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 [12, 58, 55]. While several existing clone detection algorithms target source code [61, 99, 15, 26, 24], 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.

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.

The growing popularity of machine learning-based program binary analysis. Binary code analysis (BCA) allows software engineers to directly analyze binary executables without access to source code. It is widely used in various domains where there is limited availability of source code. Today, BCA has become more important than ever due to legacy programs that have been installed in a variety of environments, including the Internet of Things (IoT).

Note that it is difficult to directly analyze binary executables when compared to program source code. First, it is challenging, if not impossible at all, to recover the original source code or semantic information from the representation of binary code.

Second, commercial software and operating systems are usually slightly obfuscated to deter reverse engineering and unlicensed use. On the other hand, system and kernel libraries are often optimized to reduce disk space requirements. That is why machine learning techniques have been widely employed on this domain, since they can automatically extract features through large amounts of data and have achieved significant success in the field of source code analysis. Currently, machine learning-based BCA has become a significant research topic in vulnerability detection, function recognition, and other areas [108].

1.3 Overview

This dissertation contains eight chapters.

- Chapter 1 motivates the need for improving the robustness and time-to-solution for software security. It also identifies the significance of having a joint statistical and formal learning approach for securing software applications.
- Chapter 2 and Chapter 3 demonstrate two novel techniques that make use of joint learning frameworks for memory protection. Specifically, *SIMBER* integrates with statistics-guided inference to remove redundant array bound checks based on runtime profile. *Clone-Hunter* uses a machine learning-based binary code clone detection to speedup the elimination of redundant array bound checks in binary executables.
- In Chapter 4 and Chapter 5, we develop a domain-specific code clone detection framework. We first provide a novel deep learning-based technique to detect pointer-related code clones in binary executables. We further demonstrate how formal method (e.g., Symbolic Execution) can be used to reduce the false positives introduced by pure machine learning-based code clone detection.
- Chapter 6 presents a novel technique that deploys a deep learning language model to reduce attack surface in legacy programs in practice.

- Chapter 7 lists some related works and how they compare to our proposed approaches.
- Chapter 8 presents the conclusions of this dissertation work and discusses future research directions.

Chapter 2 Ameliorating Memory Bound Checks through Joint Learning

In this chapter, we demonstrate SIMBER, a simple, model-based inference to identify redundant bound checks based on runtime statistics from past program executions.

2.1 Background

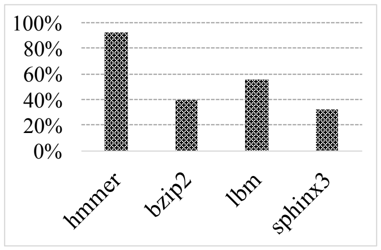


Figure 2.1: Runtime overhead for Softbound compared to original application

In order to protect software from spatial memory/array bound violations, tools such as Softbound [80] have been developed that maintains metadata such as array boundaries along with rules for metadata propagation when loading or storing pointer values. SoftBound stores the base and bound information of pointers when they are initialized, then performs the bound checks when pointers are dereferenced. For example, for an integer pointer *ptr* to an integer array *intArray*[100], SoftBound stores *ptr_base* = &*intArray*[0] and *ptr_bound* = *ptr_base* + *sizeof(intArray)*. When checking pointers before they are dereferenced, Softbound obtains the base and bound information associated with the target pointer *ptr*, and does the following: if the value of *ptr* is less than *ptr_base*, or, if *ptr+size* is larger than *ptr_bound*, then the program terminates. A disadvantage for such an approach is that, it can add performance overheads to application runtime especially due to unnecessary metadata tracking and pointer checking for benign pointers. Figure 2.1 shows the runtime overhead of SoftBound over an un-instrumented application as baseline in SPEC2006 benchmarks [1]. Existing works [18, 92] mainly analyze relationship between variables in source code, build constraint system based on static analysis and solve the system to determine redundant checks. In SIMBER, a novel framework is proposed where the redundant bound check elimination is performed with the guidance of runtime statistics. Even limited runtime statistics can be quite powerful when inferring the safety of pointer dereferences.

Consider the example shown in Figure 2.2, where $foo()$ copies the first ‘n’ characters in string *src* to *dest*, and pads each remaining position with a given pattern ‘0000’. Pointer *dest* is dereferenced in the first *for* loop as well as the *while* loop and pointer *src* is dereferenced only in the first *for* loop. Softbound checks (denoted by *CHECK_SB*) will be added before each pointer dereference, e.g., line 7, 8, 18. For every iteration of the *for* and *while* loops, the *CHECK_SB* will be executed thus producing intensive checks that could be unnecessary.

A static approach such as ABCD [18] that relies on building constraint systems at various program points can only eliminate redundant checks of $*(dest + i)$ in line 7, by identifying constraints $i < j$ (in line 16) and $j < dest\_bound$ (when checks are performed in line 18). Such a static analysis is ineffective for $*(src + i)$ and $*(dest + j)$, which cannot be removed by only constraint solving. As a result, more than half bound checks still remain in $foo()$, and bound information of both pointers needs to be kept and propagated into $foo()$ at runtime.

In this paper, we show that removing all checks in *foo*() is indeed possible using SIMBER, where the solution stems from two key observations. First, eliminating all checks in *foo*() requires only comparing the range of indices i and j with base and bound information for *dest* and *src*. In fact, function-level complete elimination of all checks in $foo()$ can be made if $i_{max} \leq dest\_bound$, $i_{max} \leq src\_bound$ and $j_{max} \leq dest\_bound$. Next, we find that indices i and j only depend on n and length of *src* respectively. Due to this positive dependency, i_{max} and j_{max} are guaranteed to decrease if n or len become smaller during a future execution. Hence, the checks are redundant for executions that have smaller n and len if larger values are already checked.

2.2 SIMBER System Design

SIMBER consists of five major modules: Dependency Graph, Statistical-guided Inference, Knowledge Base, Runtime checks removal and Check-HotSpot Identification. Figure 2.3 presents our system diagram. Given a target pointer, SIMBER

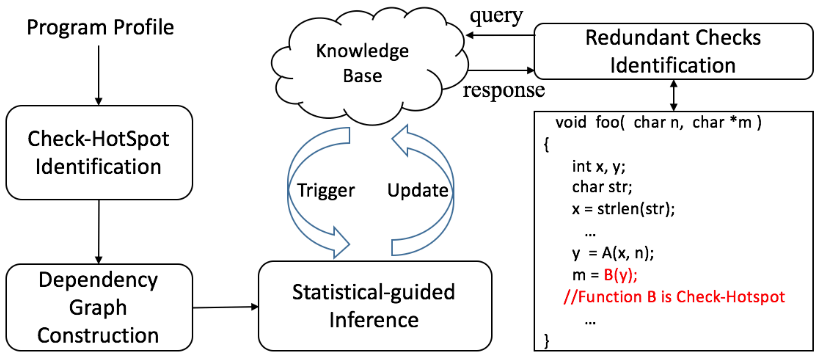
```
foo_SB
(char *dest,char *src,int n)
{
 int i, j;

 for (i=0; i<n; i++)
 {
  CHECK_SB(dest+i);
  CHECK_SB(src+i);
  *(dest+i) = *(src+i);
  //add TC1 here
 }

 int len=strlen(src);

 while (i<len)
 {
  for (j=i; j<i+4; j++)
  {
   CHECK_SB(dest+j);
   *(dest+j) = '0';
  }
  i+=4;
  //add TC2 here
 }
}
 main()
 {
   char *dest, *src;
   int n;
   ...
   foo_SB (dest, src, n);
   ...
 }
```

Figure 2.2: Softbound-instrumented source code


Program Profile
Knowledge Base
query
response
Redundant Checks Identification
Check-HotSpot Identification
Trigger
Update
Dependency Graph Construction
Statistical-guided Inference

```
void foo( char n, char *m )
{
   int x, y;
   char str;
   x = strlen(str);
    ...
   y = A(x, n);
   m = B(y);
  //Function B is Check-Hotspot
    ...
}
```

Figure 2.3: SIMBER Overview and Key Componets

aims to determine if the pointer dereference needs to be checked. SIMBER collects **pointer-affecting** variables which can affect the value of target pointers. If the data point representing pointer-affecting variables is inside the safe region inferred from previous executions, then this dereference is guaranteed to be safe.

2.2.1 Dependency Graph Construction

Dependency Graph (DG) is a bi-directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, which represent program variables as vertices in $\mathcal{V}$ and models the dependency between the variables and array indices/bounds through edges in $\mathcal{E}$. We construct a DG for each function including all if its pointers and the pointer-affecting variables that may affect the value of pointer. We add trip count(number of times a branch is taken) as auxiliary variables to assist the analysis of loops.

Definition 1 (DG-Node)**.** The nodes in dependency graphs are the variables that can affect the pointers such as a) the variables that determine the base of pointers through pointer initialization, assignment or casting; b) variables that affect the offset and bound of pointers like array index, pointer increment and variables affecting memory allocation size; c) Trip Count (TC): the number of times a branch (in which a target pointer value changes) is taken.

Definition 2 (DG-Edge)**.** DG-Node v_1 will have an out-edge to DG-Node v_2 if v_1 can affect v_2.

To construct dependency graph, we need to know the variable declaration types, statement types and the dependencies of code. Thus, we use Abstract Syntax Tree (AST) for code analysis. AST is a type of data structures used in compilers, which can be used for representing the structure of program code. We instrument AST API from Joern tool [111], a platform for static code analysis of C and C++, which can generate code property graphs, to construct AST for each function.

Algorithm 1 Dependency graph construction for a given function $foo()$

```
1: Input: source code of function $foo()$
2: Construct AST of function $foo()$
3: Initialize $\mathcal{V} = \phi$, $\mathcal{E} = \phi$
4: for each variable $v$ in AST do
5:     $\mathcal{V} = \mathcal{V} + \{v\}$
6: for each statement $s$ in AST do
7:     for each pair of variables $j, k$ in $s$ do
8:         add edge $e(j, k)$ to $\mathcal{E}$ according to Remark 2.2.1
9: Output: Dependency-Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
```

Algorithm 1 shows the pseudo code of dependency graph construction for function $foo()$. First, we obtain all pointers and their pointer-affecting variables and represent them as DG-Nodes. Second, for each pair of identified DG-Nodes, we assign a DG-Edges according to the rules in Remark 2.2.1.

We traverse dependency graph and identify adjacent DG-Nodes that represent the pointer-affecting variables associated with each target pointer. Each target pointer will have an entry in the form of $(func : ptr, var_1, var_2, ...,$
$var_n)$ where $func$ and ptr is the name of the function and pointer, respectively,

with var_i being the name of pointer-affecting variables associated with pointer ptr in function $func$. By logging the values of these variables during program executions, we next build conditions for redundant bound check elimination.

Remark. ***Edges added into Dependency Graph:***

E1 Assignment statements	A := B	$B \rightarrow A$
E2 Function parameters	Func(A,B)	$B \leftrightarrow A$
E3 Loops	for.../while...	*Add TC to Loops*
(1) Assignment inside Loops	A := B	$TC \rightarrow A$
E4 Array Indexing	A[i]	$i \rightarrow A$

2.2.2 Statistical-guided Inference

This module builds safe region based on the pointer-affecting variables identified by dependency graphs and update the safe region through statistical inference from previous execution. Once the pointer-affecting variables for the target pointer are extracted in Section 2.2.1, SIMBER collects the value of pointer-affecting variables from previous execution and produces a **data point** in Euclidean space with the coordinates of data point being the value of pointer-affecting variables. The dimension of the Euclidean space is the number of pointer-affecting variables for the target pointer. The inference about pointer safety can be derived as follows. Suppose a data point p from prior execution with pointer-affecting variables $vp_1, vp_2, ..., vp_d$, is checked and deemed as safe. Another data point q for the same target pointer but from another execution, is collected with pointer-affecting variables $vq_1, vq_2, ..., vq_d$. If each pointer-affecting variable of q is not larger than that of p, e.g., $vq_1 \leq vp_1$, $vq_2 \leq vp_2$, ..., $vq_d \leq vp_d$, then the bound checks on the target pointer can be removed in the execution represented by q.

Intuitively, positively correlated pointer-affecting variables such as array *index* are safe when they are smaller and negatively correlated pointer-affecting variables such as array *bound* are safe when they are larger. To apply such inference to negatively correlated variables, we unify the representation of pointer-affecting variables by converting the variable *bound* to *C-bound* where C is a large constant that could be the maximum value of an unsigned 32-bit integer.

Definition 3 (False Positive)**.** A false positive occurs if a bound check identified as redundant is indeed necessary and cannot be safely removed.

Definition 4 (Safe Region (SR))**.** Safe region is an area that is inferred and built from given data points, such that for any input points within the region, the corresponding target pointer is guaranteed to have only safe memory access, e.g., all bound checks related to the pointer can be removed with zero false positive, under the assumption that point-affecting variables have monotonic linear relationships with pointer bound.

Thus, the Safe Region of a single data point is the enclosed area by projecting it to each axis, which includes all input points that have smaller (pointer-affecting) variable values. For example, the safe region of a point $(3, 2)$ is all points with the first coordinate smaller than 3 and the second coordinate smaller than 2 in $\mathbb{E}^2$. We can obtain the Safe Region of multiple data points by taking the union of the safe regions generated by each data point.

Given a set $\mathcal{S}$ which consists of N data points in $\mathbb{E}^D$, where D is the dimension of data points, we first project point $s_i, i = 1, 2, ..., N$, to each axis and build N enclosed area in $\mathbb{E}^D$, e.g., building safe region for each data point. The union of these N safe regions is the safe region of $\mathcal{S}$, denoted by $SR(\mathcal{S})$. *Thus, if a new data point s_{new} falls inside $SR(\mathcal{S})$, we can find at least one existing point s_k from $\mathcal{S}$ that dominates s_{new}. That is to say, the enclosed projection area of s_k covers that of s_{new}, which means for every pointer-affecting variable, the var_i of s_k is larger than var_i of s_{new}. Hence s_{new} is guaranteed to be safe when accessing the memory. Generally, when the index/offset variables of new data points are smaller than existing data points or the bound variable of new data point is larger than existing data point, the new data point will be determined as safe.*

There are data points that can not be determined as safe or not by current safe region but later verified as legitimate. Such data points can be used to dynamically update the safe region. Given current safe region $SR(\mathcal{S})$ and the new coming data point s_{new}, $SR(\mathcal{S})$ will be updated to $SR(\mathcal{S})'$ by:

$$SR' = SR(\mathcal{S} \cup s_{new}) = SR(\mathcal{S}) \cup SR(s_{new}) = SR(\mathcal{S}) \cup \mathcal{T}, \tag{2.1}$$

where $\mathcal{T}$ is the set of safe points inside $SR(s_{new})$ but outside $SR(\mathcal{S})$. If $\mathcal{T}$ is empty which means $SR(s_{new})$ is contained by $SR(\mathcal{S})$, then there is no need to update the safe region $SR(\mathcal{S})$. Otherwise the update of safe region encapsulates two scenarios:

- There are positively correlated pointer-affecting variables (such as array index) of s_{new} that have larger values than corresponding pointer-affecting variables of all points in $SR(S)$,
- There are negatively correlated pointer-affecting variables (such as bound of pointers) of s_{new} that are smaller than those of all points in $SR(S)$

When one or both of above scenarios occur, the safe region will be enlarged to provide a higher percentage of redundant bound check elimination.

2.2.3 Knowledge Base

SIMBER stores the safe regions for target pointers in a disjoint memory space - Knowledge Base. The data in Knowledge Base, in the format of (*key, value*), represents the position and the sufficient conditions for removing the redundant bound checks for each target pointer. Statistical Inference can be *triggered* to compute the Safe Region by Knowledge Base when we detect redundant checks, then the Knowledge Base can be *updated* as more execution logs are available.

We use SQLite [2] to store our Knowledge Base. We created *RemovalCondition* table, which has fields including function names, pointer names and the corresponding conditions for redundant checks elimination.

If the data points are of one dimension, we store a threshold of pointer-affecting variable as the safe region for checks elimination. For higher dimensional data points, in case the safe region becomes too complex, we can store a pareto optimal safe region of less data points instead of the union of safe regions.

2.2.4 Runtime Bound Check Elimination

We instrument source code of benchmark programs to add *SIMBER*() function which verifies the condition of bound check elimination by comparing pointer-affecting variables collected from new executions with statistics from knowledge base before function calls. If the new data point is inside the built safe region, the propagation of bound information and the bound checks can be removed from this function.

We maintain two versions of Check-Hotspot functions: the original version (which contains no bound checks) and the softbound instrumented version that has bound checks. By choosing one of the two versions of the function to be executed based on the result of *SIMBER*() verification, we can either skip all bound checks inside the function (if the condition holds) or proceed to call the original function (if the condition is not satisfied) where bound checks would be performed as illustrated in figure 2.2. We avoid bound checks in function level by utilizing the compiling option "-blacklist" provided by softbound. We put the original names of Check-Hotspot functions in blacklist and copy each of these functions to another identical function but with a different name in the source code. *SIMBER*() verification leads to a tiny increase of code size by 1.7% since we only do runtime checks removal for Check-Hotspot functions. Also, the runtime overhead introduced by *SIMBER*() verification is small comparing to Softbound checks overhead.

2.2.5 Check-HotSpot Identification

In order to maximize the benefit of our runtime check elimination, we focus on program Check-Hotspots to minimize SIMBER overhead. Check-Hotspots are functions with high levels of pointer activity that can contribute to high overheads of bound checks.

We identify Check-HotSpots as follows: a) Program profiling: We use Perf profiling tool [33] to profile two versions of programs: non-instrumented, original version and softbound-instrumented source code. b) Function-level overhead: We compute

```
//original foo() function
foo
(char *dest,char *src,int n)
{...}

//softbound instrumented foo()
foo_SB
(char *dest,char *src,int n)
{...}
main()
{
 char *dest, *src;
 int n;
  ...
 /*determine whether it's
  inside the safe region*/
 if(SIMBER(dest,src,n))
 {
  foo(dest, src, n);
 }
 else
 {
  foo_SB (dest, src, n)
 }
 ...
}
```

Figure 2.4: SIMBER optimized code that checks if bound checks can be removed at function level

the difference in absolute execution time spent on different functions between non-instrumented source programs and softbound-instrumented programs to capture the extra time spent on Softbound-related code. For every function, we calculate the function-level overhead as the ratio of the time spent on softbound-related code to the total execution time spent in non-instrumented, original version. c) Identify Check-HotSpots: In our analysis, we list all of the functions with function-level overhead of at least 5% as the Check-HotSpots.

2.2.5.1 Example

SIMBER modifies the source code by adding two branches as shown in figure 2.4. The function *SIMBER()* verifies if the inputs satisfy the condition for check elimination and choose one of the branches accordingly.

Recall the SoftBound instrumented *foo*() function from figure 2.2 . We add trip counts $TC1$ and $TC2$ for the *for* loop in line 5 and *while* loop in line 14. According to the construction rules of dependency graph, there should be edges from node TC_1 to both pointer nodes and an edge from TC_2 to pointer node *dest*. Further, the values of TC_1 and TC_2 are determined by the exit condition of the loops they are recording, producing edges from *len* to TC_2 and from n to TC_1 in the dependency graph. The pointer-affecting variables of *dest* is $(len, n, C - dest\_bound)$. Suppose C is defined as 1024 and we have three previous runs with pointer-affecting variables as follows: $(200, 160, 1024 - 256)$, $(180, 120, 1024 - 256)$ and $(150, 140, 1024 - 512)$, denoted by

p_1, p_2, p_3 respectively. The safe region for check elimination will be built based on above three points in a $\mathbb{E}^3$ space according to the approach described in section 2.2.2.

In future executions, new data point p with pointer-affecting variables $(p_{len}, p_n,$ $p_{dest\_bound})$ are verified by $SIMBER()$ to determine if point p is inside this safe region in order to make bound check elimination decisions. As long as we can find one point from p_1, p_2 and p_3 that has pointer-affecting variables larger than those in point p, then the memory access of pointer $foo\_SB : dest$ is safe. Safe points could have but are not limited to the following pointer-affecting values: $(190, 150, 512)$, $(140, 130, 512)$.

SIMBER investigates the relationship among pointer bound and variables that can affect bound as the same in [18]. It can also be extended to analyze the base information of pointers.

2.3 Evaluation

We use Softbound as the baseline to evaluate the effectiveness of SIMBER in removing redundant bound checks. All measurements are preformed 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 select several applications from SPEC 2006 benchmark suite [1], which have high Softbound-overheads, including *bzip2*, *hmmer* from SPECint and *lbm*, *sphinx3* from SPECfp. In the evaluation, we first instrument the applications using Softbound and employ Perf to identify the Check-HotSpot functions in all applications. Similar to [18], we consider the optimization of upper- and lower-bound checks as two separate problems. In the following, we focus on eliminating redundant upper-bound checks while the dual problem of lower-bound checks can be readily solved with the same approach. For our proposed SIMBER solution, the inputs we used for testing are from the *reference* workload provided with SPEC benchmarks. In general, for applications that do not provide developer supplied representative test cases, we note that fuzzing techniques [76] [102] can be applied to generate test cases. The policies considered in our evaluation are a) Softbound instrumentation (denoted as **Softbound**). b) SIMBER Optimized Softbound with redundant bounds check removal (denoted as **S.O.S**).

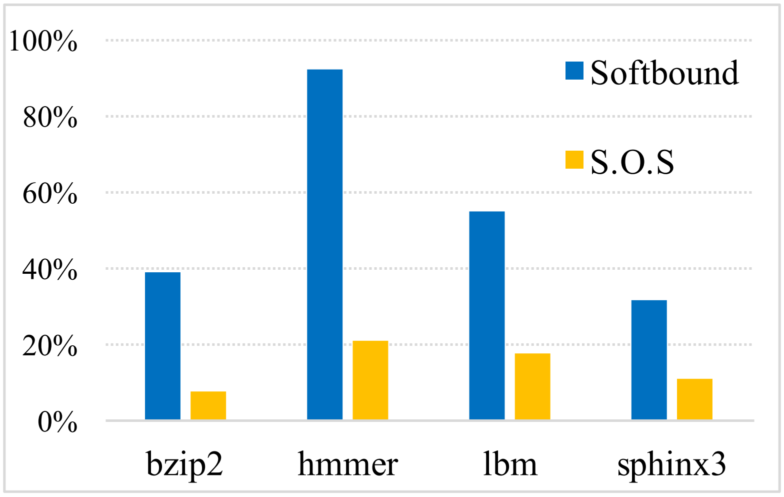


Figure 2.5: Comparison of normalized execution time overhead for Softbound and S.O.S

Our Check-HotSpot identification identifies 8 functions: *bzip2::mainGtU*, *bzip2::gen -erateMTFValues* and *bzip2::BZ2_decompress* with total 68.35% performance overhead in bzip2, *hmmer::P7Viterbi* with 98.01% performance overhead in hmmer, *lbm:: LBM_performStreamCollide* with 86.19% performance overhead in lbm and *sphinx3:: vector_gautbl_eval_logs3*, *sphinx3::mgau_eval* and *sphinx3::subvq_mgau_shortlist* with 62.58% performance overhead in sphinx3 for Softbound respectively. We note that some Check-HotSpot functions contribute to high softbound overhead mainly because they are executed frequently, e.g., *bzip2::mainGtU* is called more than 8 million times, despite having small code footprint.

2.3.1 Redundant Checks Removal Evaluation

Figure 2.5 shows the comparison of execution time overhead under Softbound and S.O.S, normalized by the execution time of original applications without performing bound checks. In particular, using the same inputs provided by SPEC benchmarks, we measure the runtime overhead for each benchmark application/Check-HotSpot functions, before and after SIMBER is enabled. Due to the ability to eliminate redundant bound checks, S.O.S. achieves significant overhead reduction. The highest reduction achieved by SIMBER is *hmmer*, with a 86.94% execution time reduction compared to Softbound. For *bzip2*, *lbm* and *sphinx3*, softbound overheads are decreased from 39% to 8%, 55% to 18%, and 31% to 11% with SIMBER overhead 4%,

Benchmark::Function Name	**Total bounds checks**	**Redundant checks removed**	**False Positive**
bzip2::generateMTFValues	2,928,640	1,440,891 (49.2%)	0 (0.0%)
bzip2::mainGtU	81,143,646	81,136,304 (99.9%)	0 (0.0%)
bzip2::BZ2_decompress	265,215	196,259 (74.0%)	0 (0.0%)
hmmer::P7Viterbi	176,000,379	124,960,267 (71.0%)	0 (0.0%)
lbm::LBM_performStreamCollide	128277886	128277886 (100.0%)	0 (0.0%)
sphinx3::vector_gautbl_eval_logs3	2,779,295	2,779,295 (100.0%)	0 (0.0%)
sphinx3::mgau_eval	725,899,332	725,899,332 (100.0%)	0 (0.0%)
sphinx3::subvq_mgau_shortlist	24,704	4,471 (18.1%)	0 (0.0%)

Table 2.1: Number of bounds checks required by Softbound and removed by SIMBER in each Check-HotSpot function

5% and 2% respectively. Overall, SIMBER achieved an average 65.31% execution time reduction for Check-HotSpots.

To illustrate SIMBER's efficiency in eliminating redundant bounds checks, Table 2.1 shows the number of total bounds checks required by softbound and the number of redundant checks removed by SIMBER along with rate of false positive reported under S.O.S. As shown in Section 2.2.2, SIMBER guarantees zero false positive, since only redundant bound checks are identified through its statistical inference and safe region construction.

2.3.2 Evaluating memory overhead and code increase in SIMBER

We note that SIMBER's memory overhead for storing Knowledge Base and additional code instrumentation are modest. The Knowledge Base mainly stores the constructed safe region. Our experiments show that the worst memory overhead is only 20KB and the maximum code size increased is less than 5% of the Check-HotSpot functions. Across all applications, SIMBER has an average 5.28KB memory overhead with an average 1.7% code increase. Overall, we reduce memory overhead by roughly 50% compared to Softbound memory requirements.

	Time spent in		
Function name	**Softbound**	**S.O.S**	**Execution time Reduction**
bzip2::generateMTFValues	77.21s	39.46s	48.89%
bzip2::mainGtU	47.94s	6.26s	86.94%
bzip2::BZ2_decompress	35.58s	9.10s	74.42%
hmmer::P7Viterbi	3701.11s	812.91s	78.04%
lbm::LBM_performStreamCollide	1201.79s	407.06s	66.13%
sphinx3::vector_gautbl_eval_logs3	1580.03s	318.10s	79.87%
sphinx3::mgau_eval	1582.68s	473.10s	70.11%
sphinx3::subvq_mgau_shortlist	270.84s	221.81s	18.1%

Table 2.2: Execution time of Check-HotSpot functions under Softbound and SIMBER, and the resulting execution time reduction.

2.3.3 Case Studies

For the Check-HotSpot functions from our SPEC applications, Table 2.2 presents the softbound overhead before and after redundant bounds checks removed by SIMBER, as well as the resulting execution time reduction.

bzip2 *bzip2* is a compression program to compress and decompress inputs files, such as TIFF image and source tar file. We identified three Check-HotSpot functions in *bzip2*: *bzip2::mainGtU*, *bzip2::generateMTFValues* and *bzip2::BZ2_decompress*. We use the function *bzip2::mainGtU* as an example to illustrate how SIMBER removes redundant checks in detail. Using Dependency Graph Construction from section 2.2.1, we first identify *nblock*, i_1, and i_2 as the pointer-affecting variables in *bzip2::mainGtU* function. For each execution, the Statistical Inference module computes and updates the Safe Region, which results in the following (sufficient) conditions for identifying redundant bounds checks in *bzip2::mainGtU*:

$$nblock > i_1 + 20 \text{ or } nblock > i_2 + 20 \tag{2.2}$$

Therefore, every time this check-hotspot function is called, SIMBER will trigger run-time checks removal if the inputs variables' values: *nblock*, i_1, and i_2 satisfy the conditions above. Because its safe region is one dimensional, the calculation of check removal conditions is indeed simple and only requires program input variables i_1 and i_2 (that are the array indexes) and *nblock* (that is the input array length). If satisfied, the conditions guarantee complete removal of bounds checks in *bzip2::mainGtU* function. Our evaluation shows that it is able to eliminate over 99% redundant checks.

For the second Check-HotSpot function *bzip2::generateMTFValue*, we apply bound -s checks removal to five different target pointers inside of the function. We observed that three out of five target pointers, with constant array length, are always safe from out of bounds accesses. Thus, the safe region only involves one single variable and SIMBER is able to get the exact removal conditions based a few data points. The other two target variables' array length are not constant, the number of dimensions increases and it is more complex for bound checks removal. In this case, SIMBER optimization reduces execution time overhead from 77.21s to 39.46s and the average execution time reduction by 48.89%. We can see this number is near proportional to the number of checks removed by SIMBER in Table 2.1.

The last Check-HotSpot function *bzip2::BZ2_decompress* has over 200 lines of code. Similar to function bzip2::generateMTFValue that it also has five target pointers under the same check elimination conditions. SIMBER deploys a function-level removal for function bzip2::BZ2_decompress. As we can see from Table 2.2, SIMBER obtained a 74.42% execution time reduction, which is consistent with the number of redundant bound checks identified by SIMBER presenting in Table 2.1.

hmmer *hmmer* is a program for searching DNA gene sequences, which implements the *Profile Hidden Markov Models* algorithms and involves many double pointers operations. There is only one Check-HotSpot function, *P7Viterbi*, which contributes over 98% of softbound overhead.

Inside of the *hmmer::P7Viterbi* function, there are four double pointers: *xmx, mmx, imx* and *dmx*. To cope with double pointers in this function, we consider the row and column array bounds separately and construct a Safe Region for each dimension. Besides the 4 double pointers, we also identify conditions for identifying redundant bound checks for another 14 one-dimensional arrays and pointers. In this case, SIMBER is able to eliminate the most redundant checks for these 14 one-dimensional arrays due to its simplicity of checks removal conditions' calculation. However, for these four double pointer, SIMBER is more conservative to ensure no false positive and the conditions calculation is more complex than one-dimensional array. Thus, the softbound overhead is significant reduced from 3701.11s to 812.91s, representing 78.94% execution time reduction.

lbm lbm is developed to simulate incompressible fluids in 3D and has only 1 Check-HotSpot function: *lbm::LBM_performStreamCollide*. The function has two pointers (as input variables) with pointer assignments and dereferencing inside of a for-loop. Thus, under Softbound it suffers from high bounds-check overhead, because pointer dereferencing occurs repeatedly inside the for-loop, resulting in frequent bound checks. Using SIMBER, we obtain the bounds check removal conditions for each pointer dereferencing. Further, combining these conditions, we observed that the pointer dereferencing always access the same memory address, implying that it is always safe to remove all bound checks in following executions after bound checks are performed and successfully passed in the first execution. Thus, SIMBER is able to eliminate 100% redundant checks with a 66.13% execution time reduction.

sphinx3 Sphinx3 is a well-known speech recognition system, it is the third version of sphinx derived from sphinx2 [52]. For the first Check-HotSpot function *sphinx3::vec -tor_gautbl_eval_logs3*, there are four target pointers inside this function. Due to the identical access pattern, once we derive the bounds check removal conditions for one single pointer, it can also be used for all the others, allowing all redundant checks to be eliminated simultaneously in this function. As shows in Table 2.1, SIMBER eliminates 100% of redundant checks with a resulting execution time of 318.10s, which achieves the optimal performance.

We observed a similar behavior for the second Check-HotSpot function *sphinx3 -::mgau_eval*. SIMBER achieves 100% redundant bounds check removal with overhead reduction of 70.11%, from 1582.68s required by softbound to 473.10s after SIMBER's redundant bound checks elimination.

The last function *sphinx3::subvq_mgau_shortlist* also has four target pointers. SIMBER optimized softbound overhead is 221.81s, compared to the original softbound overhead of 270.84s. For this function, SIMBER only removed 18.1% redundant checks, which is the lowest in our evaluations.The reason is that this function only contributes 5% softbound overhead with only a few number of redundant checks. Moreover, one of the pointers: *vqdist* inside of this function has indirect memory access that its index value is from another pointer: *map*, which means it uses a pointer's value as another pointer's offset. The dependency graph we constructed cannot represent the indirect memory access relation between these two pointers, since SIMBER is not able to remove indirect memory access pointers, it only removes about 18% bound checks. We note that capturing such memory access dependencies is possible via extending our dependency graph to model complete memory referencing relations. We will consider this as future work.

2.4 Summary

In this chapter, we propose SIMBER, a framework integrating with statistics-guided inference to remove redundant array bound checks based on runtime profile. Its statistical inference adaptively builds a knowledge base using program execution logs containing variables that affect pointer values, and then uses this information to remove redundant array bound checks inserted by popular array bound checkers such as Softbound. SIMBER reduces performance overhead of Softbound by up to 86.94%, and incurs a modest 1.7% code size increase on average to circumvent redundant bound checks inserted by Softbound. Currently, SIMBER works at function-level granularity. For future work, we will study ways to deploy SIMBER at a finer granularity to remove redundant bound checks

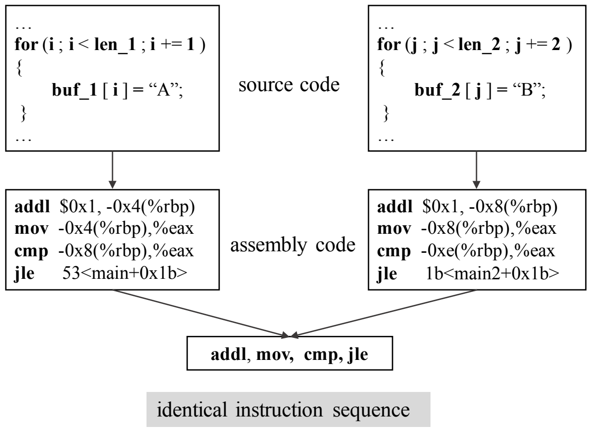
Chapter 3 Accelerating Code Analysis through Joint Learning Approach

In this chapter, we design Clone-hunter, a practical and scalable framework for redundant bound check elimination in binary executables.

3.1 Background

Our work is motivated by the key observation that software applications usually have an abundant number of similar code fragments, called *code clones* [58, 59]. Two code fragments can be named as *code clones* if they are similar to each other based on a given code similarity matrix (e.g., tree-based code similarity [55]). *There is a high possibility that if checking array bounds is deemed redundant for a certain code fragment, it can also be removed from its corresponding code clones.* Effectively, instead of analyzing every single pointer, we leverage binary code clone detection techniques and reduce the time-to-solution in terms of eliminating redundant bound checks in binaries.

Clone-Hunter performs rapid elimination of redundant bound checks in binary applications through identifying code clones, and forming clusters of such clones, we pick random seed samples from each cluster, and with the help of a binary symbolic executor, determine whether bound checks are necessary on the seed. If deemed unnecessary, the decision to remove bound checks is replicated to all of the other clone samples, thereby significantly speeding up the redundant bound check elimination process. We improve the confidence of our decision to replicate bound check removal through performing random spot-checks. That is, we randomly select a group of clones within each cluster and determine whether bound checks can be removed through symbolic execution. This verifies the soundness of our decision to remove bound checks in the clone samples within the cluster. Our experimental results show that our approach is powerful, and can significantly reduce the performance overheads in eliminating redundant bound checks by up to 45.54% in binary applications.



```
...
for (i ; i < len_1 ; i += 1 )
{
      buf_1 [ i ] = "A";
 }
...
```

```
...
for (j ; j < len_2 ; j += 2 )
{
      buf_2 [ j ] = "B";
 }
...
```

```
addl  $0x1, -0x4(%rbp)
mov  -0x4(%rbp),%eax
cmp  -0x8(%rbp),%eax
jle     53<main+0x1b>
```

```
addl  $0x1, -0x8(%rbp)
mov  -0x8(%rbp),%eax
cmp  -0xe(%rbp),%eax
jle     1b<main2+0x1b>
```

```
addl, mov,  cmp, jle
```

Figure 3.1: Motivation example for code normalization

We note that Clone-Hunter presents a new approach that combines statistical methods (such as machine learning to identify code clones) and formal analysis tools (such as symbolic execution) to preserve array bound checks where necessary, while eliminating a vast majority of redundant checks. To the best of our knowledge, Clone-Hunter is the first proposed framework for redundant bound check elimination in application binaries. This work is significant because most of the critical binary applications deployed in military and financial domains need effective memory safety, but should not be adversely affected by the unnecessary performance overheads imposed by redundant checks[30].

3.2 Clone-Hunter System Design

3.2.1 Binary Code Clone Detection

Clone-Hunter accelerates redundant bound checks removal by identifying binary code clones, and replicates the decision to perform removal of bound checks on the corresponding code clones.

Vector Embedding: We first disassemble the target binaries, and detect code clones in the assembly code, which are functionally similar. Note that every machine instruction in binary executables is a combination of instruction type and the corresponding operands, such as memory references, registers and immediate values. Two code samples are considered as code clones if they can be deemed functionally similar except for some certain constant values, offsets in memory locations, or addresses used as branch targets. For example, Figure 3.1 shows two functionally identical source code snippets and their corresponding assembly code. As we can see, their assembly codes share the same instruction sequence but different operands. We perform *normalization* to abstract out specific addresses and register names, while preserving the instruction patterns and the logical functionality of the code regions. This enables more effective gathering of functionally similar code snippets using clustering algorithms.

```
mov  %r10,%rdi                  mov  REG, REG
sub  %eax,%r9d                  sub  REG, REG
mov  $0x1,%esi                  mov  VAL, REG
mov  %r8, %rsp                  mov  REG, REG

   Original Code                 Normalized Code
```

Figure 3.2: An example illustrating normalization for given a given binary code.

We use a sliding window method to select different code regions for code clone analysis. The method has 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. For each code region, normalization is performed, since two code regions that are syntactically or semantically equivalent may have identical instruction patterns, but may have different memory references, registers or constants. Specifically, we use an abstract operand format with three symbols, namely $\{MEM, REG, VAL\}$. Memory references are replaced by symbol MEM, register names by symbol REG or constant values by symbol VAL. Figure 2.2 shows an example how we normalize the instructions for a given code region.

Next, we cluster these normalized code regions and identify code clones via machine learning algorithms. The code regions are embedded into a feature vector space. In particular, we count the number of occurrences of assembly instructions in each code region after normalization. Let n be the total number of distinct normalized instructions. The occurrences of different instructions are collectively stored in a fea-

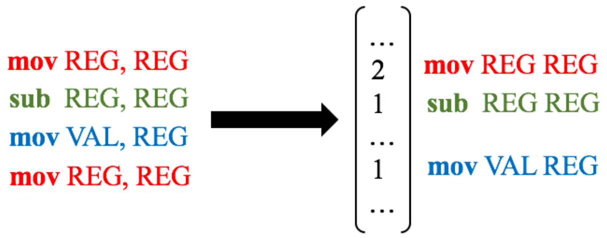


Figure 3.3: Normalized assembly code embedded into vector space

ture vector, denoted as $C_i = (C_{i_1}, C_{i_2}, ..., C_{i_n})$, where C_{i_k} (for $k = 1, \ldots, n$) measures the occurrence of normalized instruction k in code region i. This process is illustrated in Figure 3.3 for the code region example shown in Figure 3.2.

In Clone-Hunter, we employ IDA Pro binary disassembler [3] and implement instruction normalization and vector embedding in Python. The actual instruction addresses, register names prior to normalization, and code region's starting and ending addresses are stored as a query table using SQLite database [2]. This is done to reverse map normalized code samples back to the binary such that the decision of removing bound checks can be verified (Section 3.2.2).

Machine Learning-based Clone Detector: After embedding the code regions into feature vectors, we make use of the Affinity Propagation (AP) clustering algorithm for binary code clone detection. AP clustering is able to determine the number of clusters among the data points without any a priori knowledge. The embedded vectors corresponding to different code regions, $C_1, C_2, \ldots, C_m$ are referred to as m different data points in the clustering algorithm.

AP performs an iterative procedure to update the association between data points and candidate cluster centers. Let S be a similarity matrix. Its off-diagonal components $S(i, j)$ for $i \neq j$ quantify the similarity between two distinct data points, C_i and C_j, represented as the negated value of the squared euclidean distance.

$$S(i, j) = -||C_i - C_j||_2^2. \tag{3.1}$$

where $|| \cdot ||_2$ denotes the L-2 vector norm. On the other hand, the diagonal values

$S(k,k)$ are input parameters (known as the *preference*) reflecting the likelihood of data point k being chosen as a cluster center. It is easy to see that if $S(i,j) > S(i,k)$, then C_i is closer to C_j than C_k.

In the AP algorithm, there are two matrices, *Responsibility matrix* and *Availability matrix*, being updated in each iteration. In particular, $R(i,k)$ measures how well data point C_k is suited to serve as a candidate cluster center for point C_i, while $A(i,k)$ reflects how appropriate it is for C_i to choose C_k as its cluster center. The AP algorithm initializes both matrix to zero, and in each iteration, updates both $R(i,k)$ and $A(i,k)$ in a coupled fashion, according to

$$R(i,k) = S(i,k) - \max_{k':k' \neq k} \{A(i,k') + S(i,k')\} \tag{3.2}$$

$$A(i,k) = \min\{0,\ R(k,k) + \sum_{i' \notin \{i,k\}} \max\{0,\ R(i',k)\}\} \tag{3.3}$$

Note that the $A(i,k)$ is non-positive due to Equation (3.3). It is updated by the $R(k,k)$ (measuring the preference for point C_k to serve as a cluster center), plus the aggregate responsibility points that C_k receives from all other data points (reflecting its overall popularity as a cluster center among other points). The self-availability $A(k,k)$ is updated differently, i.e., $A(k,k) \leftarrow \sum_{i' \notin \{i,k\}} max\{0, R(i',k)\}$, without depending on the self-responsibility $R(k,k)$. Finally, the iterations are terminated when the changes of availabilities and responsibilities are smaller than a pre-defined threshold, implying that the cluster assignments have become stable.

We implement our clustering-based code clone detector in Python using a machine learning tool Scikit-learn [86]. We instrument its AP clustering API - *sklearn.cluster* for our clustering module.

Consolidation of Code Clones: Code Clone Consolidation removes duplicate and pointer-irrelevant code clones from further consideration. Non-pointer related clones are not useful in removal of array bound check conditions, and hence are not considered useful in our study.

We first filter out the pointer-irrelevant code clones by checking if they contain bound check-related instructions. For example, Softbound-instrumen

-ted bound checks instructions will contain "*softbound_spatial_checks*" symbol in binary executables. This enables filtering out these instructions using such symbols.

Also, as described in Section 3.2.1, we use a sliding window based analysis approach for code clone detection. We note that this can create overlapping windows resulting in partially overlapping or even duplicated code clones. To address this problem, we consolidate the code clones by computing the union of overlapping code clones, i.e., the union of their start and end instruction addresses in assembly code. Each code clone sample is denoted as a vector (s, e) where s is the starting address and e is the ending address in the code region. Two code clones, (s, e) and (s', e'), are overlapping if they have non-empty intersection, i.e., $(s, e) \cap (s', e') \neq \phi$. Thus, we use their union to consolidate them and define a maximum-sized, continuous code snippet, $(s, e) \cup (s', e')$. This consolidation procedure is performed until all consolidated code clones are non-overlapping.

We implement our Code Clone Consolidation module using Python embedded into ML-Clone Detector.

3.2.2 Symbolic Execution for Bound Verification

Clone-Hunter utilizes clustering algorithms in Machine Learning to identify binary code clones, that can be used to assist removal of redundant array bound checks. Based on our observations from a large number of code samples, it is highly likely that the redundant bound checks in two code samples can be both removed if they are functionally equivalent code clones. To *formally check* if the code clones detected by Clone-Hunter can safely remove array bound checks, we utilize binary symbolic execution as our verification tool.

There are three major steps for bound check verification and elimination in Clone-Hunter:

1. **Identification of redundant bound checks:** First, we pick a random code clone sample as *seed clone sample* in each cluster. We determine the pointer dereference is safe, and that no memory violation can exist. We deploy binary symbolic execution to execute the *seed clone sample* and check whether the

array bound checks are redundant based on the output from symbolic execution. We perform partial symbolic execution starting from beginning to end of the seed clone sample based on its instruction addresses. To deal with possibly incomplete program state while performing partial symbolic execution, we make the values of unknown variables in this code region as symbolic variables. If the pointers in *seed clone sample* turn out to be safe, then array bound checks in the corresponding code snippet may be safely removed. If not, we terminate the array bound verification procedure and apply the final decision as 'Not redundant' to the other code clones in the cluster. That is, the array bound checker tool-inserted code is kept intact and are not removed.

2. **Verification of bound identification:** Clustering algorithm cannot offer any guarantees in terms of ensuring safe bound check removal from all detected code clones. It is possible that two code snippets are found to be code clones, but have different bound safety conditions and do not allow simultaneous bound checks removal. To further improve the accuracy of Clone-Hunter, we select a random set of code clone samples within the same cluster and perform binary symbolic execution to check whether the bound checks removal conditions on these code clones are indeed similar.

3. **Applying decision to remove bound checks:** If the random code clones samples turn out to be safe, we apply the final decision as 'Redundant' to all of the code clones within the cluster, and remove the corresponding array bound checking code inserted by the memory safety tool. On the flip side, if the safety checks by symbolic executor on random code clones samples fail, then we apply the final decision as 'Not redundant' to all of the clone samples in the cluster. That is, the array bound checker tool-inserted code is kept intact and are not removed.

We instrument a binary analysis framework angr [87] for bound verification. We deploy the binary symbolic executor in angr for a target location to start performing symbolic execution in binary executables, beginning with the starting address and execute instructions within the specific code region.

3.2.3 Removal of Redundant Bound Checks

To delete instructions in binary executables, we deploy a Static Binary Rewriting tool Dyninst [90]. As we discussed earlier, we store additional information for each code region including their start and end addresses. We use this information to rewrite control transfers. We implement our Bound Check Remover in C++ with Dyninst. Given a code clone as input, we scan each instruction and remove redundant array bound checks. We obtain optimized binaries as output.

3.3 Evaluation

We provide an overview of our experimental setup, and later present our evaluation results.

3.3.1 Experiment Setup

We selected 4 different real-world applications: bzip2, hmmer, lbm and sphinx3 from SPEC2006 benchmark suite [1] and use the largest *reference* input sets to perform our study. 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 evaluate the performance of Clone-Hunter, we deploy a runtime bound checker tool: Softbound [81] that inserts array bound checks into application's binary executable files.

Bench.	**#Total Static Instructions**	**#Clusters**	**#Cloned Instructions**	**% Instructions inside clones**
bzip2	14,293	213	4,397	30.76%
sphinx3	203,708	2,771	89,647	44.01%
lbm	2,360	58	712	30.17 %
hmmer	171,376	1,440	69,324	40.45%

Table 3.1: Binary Code Clone Statistics

Bench.	**Type**	**Size** (Byte)	**PSE** **Whole Program** (sec)	**PSE** **Function-Level** (sec)	**Clone-Hunter assisted SE** (sec)
bzip2	File Compression	305K	TIME OUT	383.40	153.98
sphinx3	Speech Recognition	1.3M	TIME OUT	14010.00	6144.30
lbm	Computational Fluid Dynmaics	55K	35032.54	1584.40	387.90
hmmer	DNA Sequence Search	974K	TIME OUT	6733.28	957.36

Table 3.2: Comparison of time spent in Clone-Hunter and PSE (Pure Symoblic Execution), where SE stands for Symbolic Execution time

3.3.2 Effectiveness of Binary Code Clone Detection

We evaluated our binary code clone detector with different window sizes and stride values. The number of cloned instructions shows the pervasive presence of code clones throughout the entire program in certain applications. In general, we observed that there are more code clone samples detected with smaller window sizes. In particular, our experiments showed that we are able to detect the most code clones with maximum window size equals to 100 tokens (minimum window size = 2 tokens) and stride value of 4. Table 3.1 shows, for each benchmark, the statistics about the number of static instructions, clone clusters, number of instructions in clone samples, and the overall percentage of program instructions they represent.

As we can see, sphinx3 has the highest coverage of cloned instructions with over 44% and also has the most number of clusters generated from our machine learning algorithm.

3.3.3 Overhead of Binary Symbolic Execution

We evaluated the overhead of binary symbolic executors for checking redundancy of array bound checks using Clone-Hunter, and compared the execution time with

Pure Symbolic Execution over entire binary programs. Our baseline is the binary analysis framework angr [87].

Table 3.2 presents the runtime overhead due to pure symbolic execution and Clone-Hunter. We evaluate pure symbolic execution overhead on the entire program and conduct partial symbolic execution on each function as function-level overhead. We set up 43,200 seconds (12 hours) as TIME OUT.

In our experiments, we set up a threshold for number of code samples used for bound verification. Since the smallest cluster contains only 2 code clone samples, we chose a lower bound as 2 code clone samples. For larger clusters, we pick 30% sampling rate as upper bound to randomly select code clone samples for spot checks described in Section 3.2.2. We note that the sampling rate within the cluster is tunable depending on the user's needs. The time spent in Clone-Hunter assisted Symbolic Execution is calculated as the summation of symbolic execution times in the random seed clones within each cluster. We observe that Clone-Hunter always spends less time than angr in terms of performance overhead. Notably, angr fails to finish symbolic execution for bzip2, sphinx3 and hmmer, angr and results in TIME OUT. The time-to-solution (the time spent to remove bound checks) for Clone-hunter is 90$\times$ faster compared to pure Binary Symbolic Execution in lbm. On the other hand, it is easy to see that why pure symbolic execution takes more time in Clone-Hunter in sphinx3. Our code clone detector detected the number of clusters as 2,771, which means we need to pick at least 5,542 code clones for bound verification. On the other hand, we only need to pick at least 116 code clones in lbm.

As expected, pure symbolic execution over the entire program results in much higher runtime for angr, and often results in TIME OUT due to path explosion where every single program path needs to be explored by the symbolic executor. Some functions in bzip2 contain more loop operations and function calls, and leads to a longer symbolic execution time for entire program analysis.

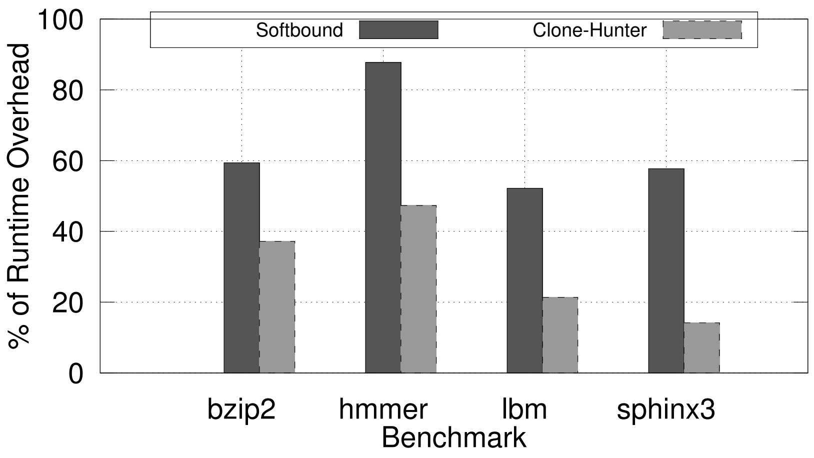


Figure 3.4: Runtime overhead of softbound-instrumented applications and Clone-Hunter. The baseline is non-instrumented applications.

3.3.4 Redundant Bound Checks Elimination

Figure 3.4 shows the comparison of Softbound's runtime execution overhead before and after using Clone-Hunter (that eliminates redundant bound checks and the overheads associated with them). Our results show that Clone-Hunter is able to significantly reduce the runtime overheads caused by redundant array bound checks in certain applications such as sphinx3 and lbm to about 20% or less. In other applications with high runtime overheads, such as hmmer, we observe about 50% reduction in execution time penalty due to Softbound checks. Clone-Hunter achieves an average reduction of 34.24% compared to Softbound runtime overheads.

We further evaluate the percentage of false positives in removing redundant bound checks. We note that a false positive occurs if a bound check is deemed redundant by Clone-Hunter, but is indeed necessary and cannot be safely removed in reality. On the flip side, false negatives occur if a bound check is deemed not redundant by Clone-Hunter but is actually unnecessary. We note that false negatives aren't security critical and only results in actually redundant checks being missed by Clone-Hunter. Therefore, we do not evaluate Clone-Hunter for false negatives in our study.

Table 3.3 shows the percentage of dynamic bound checks eliminated in all 4 benchmarks, and we observe zero false positives under Clone-Hunter. Clone-Hunter shows

Benchmark	bzip2	hmmer	lbm	sphinx3
%Dynamic Checks Removed	26.72%	42.31%	30.90%	45.54%
%False Positive Rate	0.00%	0.00%	0.00%	0.00%

Table 3.3: Percentage of Softbound's dynamic array bound checks removed by Clone-Hunter

an average 36.37% redundant bound checks elimination ratio, with the highest 45.54% at sphinx3. Other source code-based redundant bound check elimination approaches, such as SIMBER [107], report function-level statistics on how many redundant checks were eliminated. We note that exhaustive analysis of every pointer dereference is still needed, and may involve high runtime overheads in pointer-intensive applications. *To the best of our knowledge, Clone-Hunter is the first framework for removing redundant array bound checks in binary applications using a scalable machine learning-based approach.*

Note that the percentage of dynamic checks removed by our approach is not linearly related to runtime overhead. To explain this, we further analyzed the breakdown of Softbound's execution time. We note that bound checks on load instruction de-reference has $4\times$ higher runtime penalty compared to the corresponding store instruction de-reference check. This is because load instructions are on the critical path affecting program runtime directly, while store instructions are usually issued and the processor begins fetching the subsequent instruction even before stores complete. This is the reason why we observe a better reduction in Softbound overheads if we remove more load instruction de-reference checks. As we can see, sphinx3 achieves the highest reduction in performance overheads than others. We further analyzed sphinx3, and found that Clone-Hunter removes 62.33% load instruction related checks. Some functions in lbm are written with a bunch of macro functions within a user defined switch loop structure. This makes it more burdensome for the source code-based analyzers, such as SIMBER [107], to expand such macros and unroll the loops within them.

3.4 Summary

In this chapter, we presented a novel framework, Clone-Hunter, that integrates a machine learning based binary code clone detection to speedup elimination of redundant array bound checks in binary executables. We evaluated our approach using real-world applications from SPEC 2006 benchmark suite. Our results show the time-to-solution (the time spent to remove bound checks) for Clone-Hunter is 90$\times$ faster compared to pure Binary Symbolic Execution while three out of four applications fail to finish the execution.

As future work, we plan to explore better ways of finding semantic equivalence between code clones, and improve the redundant bound check removal capability of our framework.

Chapter 4 Domain Specific Code Clone Detection

In this chapter, we introduce Clone-Slicer to identify domain-specific binary code clones (e.g., pointer-related code) through program slicing. Our approach first eliminates nondomain-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.

First, we introduce the definition of code clones and related background.

4.1 Background

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 [78]; (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 [55]. Some existing code clone detection techniques apply simple pattern matching (e.g., token-based code clone detection approach [10, 58, 71]) and leverage a code similarity metric to measure the amount of similarity between two code samples.

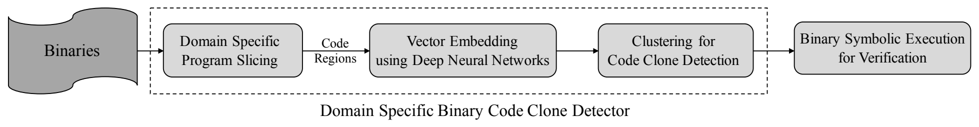

Binaries
Domain Specific Program Slicing
Code Regions
Vector Embedding using Deep Neural Networks
Clustering for Code Clone Detection
Binary Symbolic Execution for Verification
Domain Specific Binary Code Clone Detector

Figure 4.1: The Kernel of Clone-Slicer

4.2 Clone-Slicer System Design

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 4.1.

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 4.2.1). 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 4.2.2). 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 4.2.3). 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 4.2.4 describes our implementation in more detail.

4.2.1 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.

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 [27, 96, 97]. 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 [90]. We instrumented a binary analysis framework angr [87] and develop a python script to construct CFG and DDG in binaries.

4.2.2 Vector Embedding using Deep Neural Networks

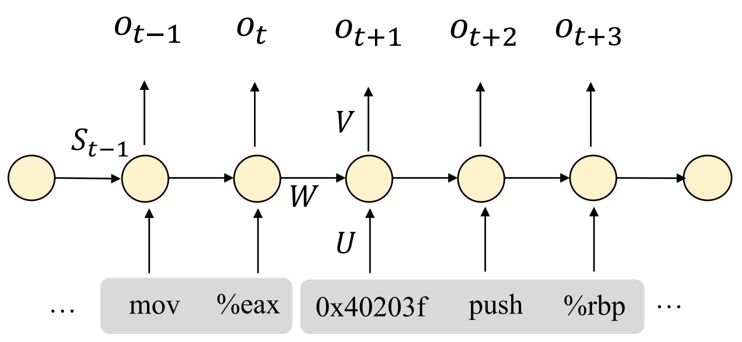


Figure 4.2: An illustration of RNN. The input of each node is a one-hot vector representing the current term in the disassembly code corpus, and output is a probability distribution predicting the next term. U, V, W are the parameters in the network, and s_t is the hidden layer state vector.

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 4.2 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 [94] 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:

$$s_t = f(Ux_t + Ws_{t-1}) \tag{4.1}$$

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 [49]. It is calculated based on current state vector along with another shared parameter $V \in \mathbb{R}^{m\times n}$, i.e., :

$$O_t = softmax(Vs_t) \tag{4.2}$$

The parameters $\{U, V, W\}$ are trained using back propagation through time (BPTT) method in our RNN network [17]. Once RNN training is complete, each term in the code corpus will have an unique embedding U from Equation (4.1), which comprises its semantic representation cross the corpus [9]. 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 4.3, 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:

$$Z(x_1, x_2) = f(W_1[x_1; x_2] + b_1), \tag{4.3}$$

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:

$$O[x_1; x_2] = g(W_2[x_1; x_2] + b_2), \tag{4.4}$$

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.,

$$E([x_1; x_2]; \theta) = ||[x_1; x_2] - O[x_1; x_2]||_2^2 \tag{4.5}$$

For a given code region with multiple terms and instructions, we adopt a greedy method [101] 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 4.3 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

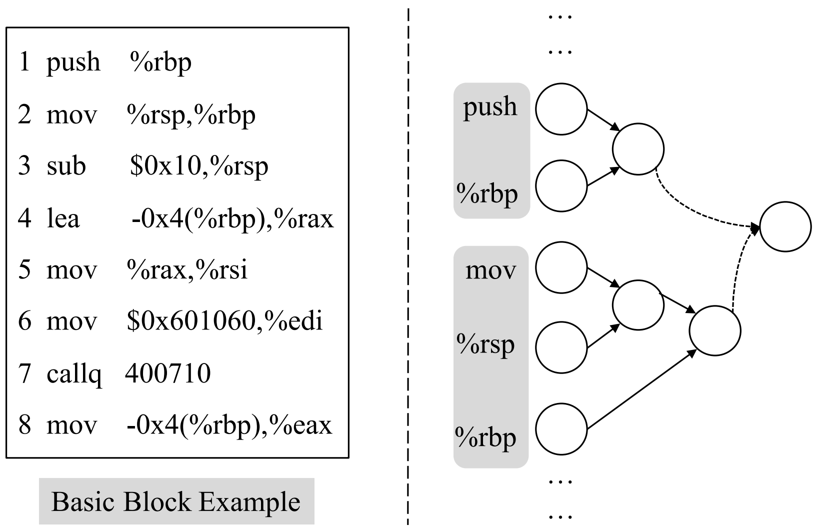

```
push    %rbp
mov     %rsp,%rbp
sub     $0x10,%rsp
lea     -0x4(%rbp),%rax
mov     %rax,%rsi
mov     $0x601060,%edi
callq   400710
mov     -0x4(%rbp),%eax
```

Figure 4.3: RAE combines embeddings from different terms and instructions through a Greedy method.

signature vector) for the instruction sequences of the given execution path.

We used IDA Pro [3] for disassembly and implemented RNN and RAE in python based on the framework proposed in [68]. For RNN, we develop a python script to tokenize the disassembly code and use the RNNLM Toolkit [77] to train RNN for each program, with the hidden layer size equal to 500.

4.2.3 Clustering for Code Clone Detection

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

Definition 5. 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:

$$S(T_1, T_2) = \frac{2S}{2S + L + R} \tag{4.6}$$

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.

Clone Detection: Given a group of feature vectors, we utilize Locality Sensitive

Hashing (LSH) [32] 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 [55, 46]. 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:

$$E([V_i; V_j]) = ||V_i - V_j||_2^2 \tag{4.7}$$

$$H([V_i; V_j]) = ||V_i - V_j||_1 \tag{4.8}$$

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:

$$E([V_i; V_j]) \geq \sqrt{H([V_i; V_j])} \approx \sqrt{L + R} \tag{4.9}$$

Based on the definition from Equation 5, 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:

$$T = \sqrt{2(1 - S) \times min(S(C_i), S(C_j))} \tag{4.10}$$

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

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.

4.2.4 Binary Symbolic Execution for Verification

Clone-Slicer makes use of clustering algorithms to identify binary code clones. In prior work Clone-Hunter [109] , 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 4.3.4)

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 [87] 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.

4.3 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 [109].

Bench.	**Size (Byte)**	**PSE (sec)**	**Clone-Hunter SE(sec)**	**Clone-Slicer SE(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 4.1: Comparison of execution time spent in Pure Symbolic Execution, Clone-Hunter and Clone-Slicer, where PSE stands for Pure Symbolic Execution time and SE stands for Symbolic Execution time

4.3.1 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].

4.3.2 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 4.4 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.

Benchmark	**#Code Clones**		**%Improvement**
	Clone-Hunter	**Clone-Hunter**	
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-Hunter**	
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, $S = 0.90$

Table 4.4: Comparison of number of code clones detected by Clone-Hunter and Clone-Hunter

4.3.3 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 [87].

For a fair comparison, we set up the same threshold for number of code samples used for verification as mentioned in Clone-Hunter, 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 4.2.4. Table 4.1 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.

4.3.4 Case Study: Removing Redundant Array Bound Checks

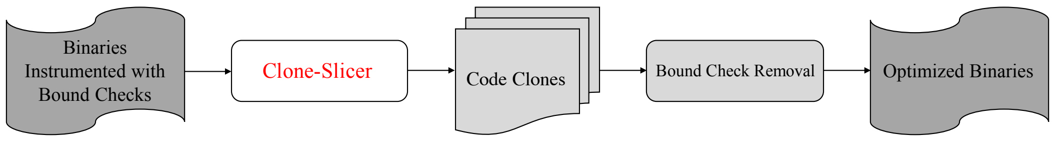


Figure 4.4: Application of Clone-Slicer kernel to remove redundant bound checks

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 4.4 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 [90] to remove bound checks in binaries.

To evaluate the performance of Clone-Slicer, we employ a runtime bound checker tool: Softbound [81] to insert bound checks in the benchmarks. Figure 4.5 shows the

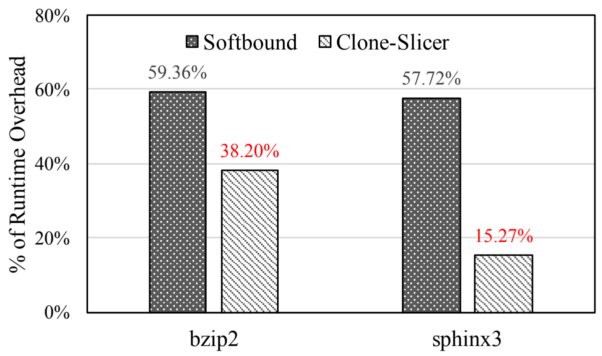


Figure 4.5: Runtime overhead of softbound-instrumented applications and Clone-Slicer. The baseline is non-instrumented applications.

comparison of Softbound's runtime execution overhead before and after using Clone-Slicer. Our results show that Clone-Slicer is 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.

4.4 Summary

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.

Chapter 5 Joint Learning Approach for Robust Clone Detection

In this chapter, we develop a machine learning and symbolic execution integrated reasoning engine, Twin-finder, to detect pointer-related code clones in source code.

5.1 Background

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 the 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 5.1. 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.

5.1.1 Motivating Example

We use real-world false positive and true positive samples in sphinx3 from SPEC20 -06 benchmark reported from a tree-based code clone detector DECKARD [55] as motiving examples. First, we give the formal definition of false positive which is defined in Definition 6.

Definition 6. 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

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

```
void dict2pid_dump (...){
...
for (i = 0; i < mdef->n_sseq; i++) {
  fprintf (fp, "%5d ", i);
  for (j = 0; j < mdef_n_emit_state(mdef); j++)
    fprintf (fp, "%5d", mdef->sseq[i][j]);
..
}
```

Pointer $\{mdef- > sseq\}$ is intertwined inside of the function

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

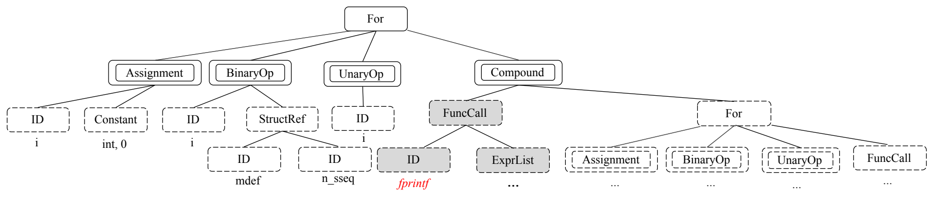
Pointer $\{gs- > codeword\}$ is intertwined inside of the function

Figure 5.1: A true positive example

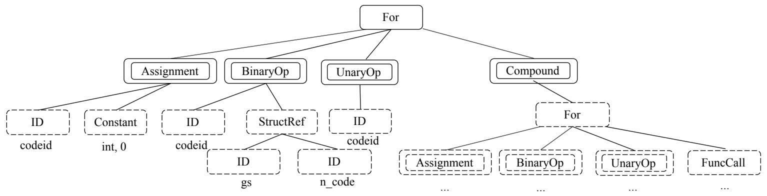
samples share different bound safety constraints in terms of pointer analysis.

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 5.1, 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 5.2 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 deep neural network (DNN)-based approach. Unfortunately, they still have inevitable drawbacks. First, given two pieces of code that differ in only a few statements but with a 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



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



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

Figure 5.2: ASTs generated from the true positive example in Figure 5.1, where the shady nodes represent the different nodes between two trees

simply derived as these two equations.

$$\{i < length(mdef- > sseq)\} \wedge \{j < length(*mdef- > sseq)\} \tag{5.1}$$

$$\{codeid < length(gs- > codeword)\} \wedge \{cid < length(*gs- > codeword)\} \tag{5.2}$$

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 5.3 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 under a certain code similarity threshold. Figure 5.4 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,

```
int32 mgau_eval (..., int32 *active)
{
...
for (j = 0; active[j] >= 0; j++)
{
  c = active[j];
...
}
```

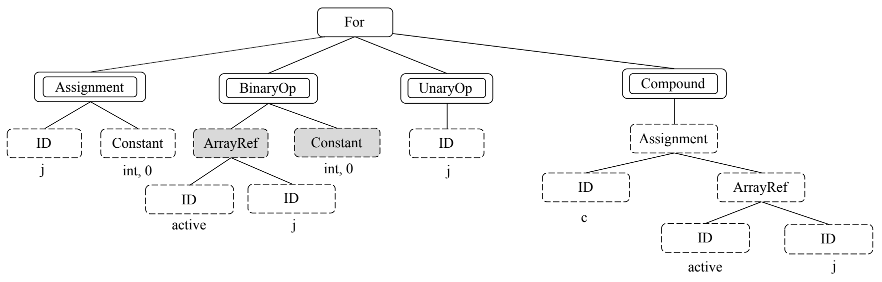
```
void lextree_hmm_histbin (lextree_t
    *lextree ,...)
{
...
for (i = 0; i < lextree->n_active; i
    ++)
{
  ln = list[i];
...
}
```

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

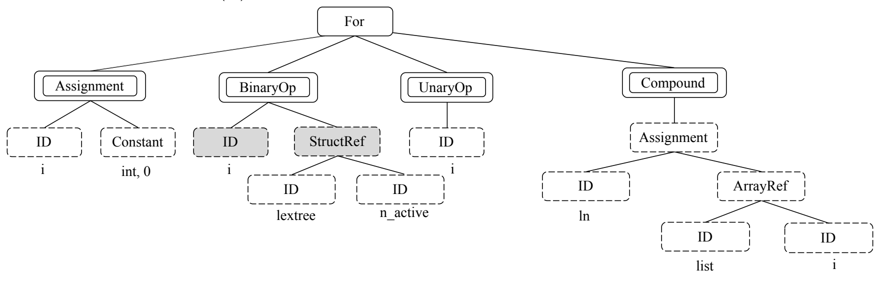
```
void fe_spec_magnitude(double *data,
     int32 data_len, double *spec,
    int32 fftsize)
{
...
IN = (complex *) calloc(fftsize,
    sizeof(complex));
...
for (wrap=0; j<data_len; wrap++,j++)
     {
  IN[wrap].r += data[j];
  IN[wrap].i += 0.0;
}
...
for (j=0; j<fftsize;j++) {
  IN[j].r = data[j];
  IN[j].i = 0.0;
}
...
}
```

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

Figure 5.3: False positive examples from sphinx3



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



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

Figure 5.4: ASTs generated from the firste false positive example in Figure 5.3, where the shady nodes represent the different nodes between two trees

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 6, 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 (line 6-9) and $\{j, fftsize\}$ are related to second *for* loop (line 11-14), 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 5.4 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.

5.2 Twin-Finder System Design

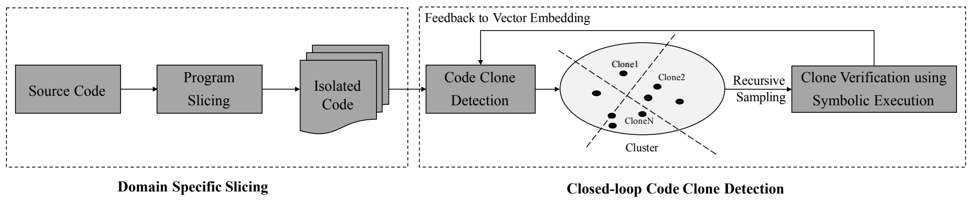


Figure 5.5: Twin-Finder Overview

In this section, we present details of our Twin-Finder and show how our system is designed. Two main components of Twin-Finder are shown in Figure 5.5, namely Domain Specific Slicing and Closed-loop Code Clone Detection.

5.2.1 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 three steps:

1. **Pointer Selection.** Given a source code of a program, we utilize the static code analysis to select all the pointers and collect related information from the code, including variable name, pointer declaration type (e.g. global variables, local variables or structures) and the location in the code (defined and used in which function). In particular, we use a program parser ANTLR[85] and a static code analysis tool Joren [111] 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 $V = \{v_1, v_2, \dots, v_n\}$ 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 addition, we also need to preserve the locations (e.g. line of code) of any selected statements in the original source code for further analysis.

5.2.2 Code Clone Detection

Twin-Finder leverages a tree-based code clone detection approach, which is originally proposed by Jiang et al. [55]. 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 4.2.1 and generate AST correspondingly. Here, we adopt the notions of code similarity, feature vectors and other related definitions from previous works [16, 55]. We deploy such method on the top of our domain specific slicing module to only detect code clones among pointer isolated codes.

Given a group of feature vectors, we utilize Locality Sensitive Hashing (LSH) [32] 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 [55, 46]. 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, which we have described all the details of such clone detection process in Section 4.2.3

5.2.3 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.

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.

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 7. 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

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

Give a clone pair sampled from the previous step, we perform symbolic execution from beginning to the end of clone samples in original source code based on the locations information (line numbers of code). The symbolic executor is used to explore all the possible paths existing in the code fragment. 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 7) 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 [20], use SMT-Lib string constraints format with BitVector theory [14, 45]. 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:

- **Matching the Variables**: To verify if two sets of constraints are equal, we omit the difference of variable names. However, we need to match the variables between two constraints based on their dependency of target pointers. For instance, two pointers dereferences $a[i] =' A'$ 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 $(z < x < n)$.

- **Checking the Equivalence**:To prove two sets of constraints $S_1 == S_2$, we only need to prove the negation of $S_1 == S_2$ is unsatisfiable.

Example. Assuming we have two sets of constraints, $S_1 = (x1 \geq 4) \wedge (x2 \geq 5)$ and $S_2 = (x3 \geq 4) \wedge (x4 \geq 5)$, where $x1$ is equivalent to $x2$ and x_3 is equivalent to x_4. We then can solve that $Not(S_1 == S_2)$ is unsatisfiable. Thus, $S_1 == S_2$.

5.2.4 Formal Feedback to Vector Embedding

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

Algorithm 2 Algorithm for Feedback to Vector Embedding

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

false positives. Since the code clone detection is based on the euclidean distance between data points over a code similarity threshold, the feedback is a mechanism to tune the feature vectors weights. Based on the constraints we obtained from symbolic execution, we are able to determine which type of variables or statements causing different constraints between two clone samples. We use such information to guide feedback to vector embedding in clone detection module. Now we describe a feedback mechanism to vector embedding in code clone detection if we observe false positives verified through the execution in Section 5.2.3.

The general idea of our feedback is that we analyze the difference between two ASTs by comparing two trees and find the differences between them. 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 3 shows the steps of feedback in detail. Given a code similarity threshold S, It takes two clone samples (C_i, C_j), corresponding AST sub-trees (S_i, S_j) and feature vectors (V_i, V_j) representing two code clones as inputs (line 1-4 in Algorithm 2), 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 clones in the future.

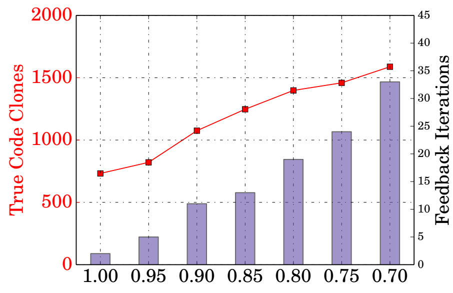
(2) If they have different constraints, we obtain the uncommon subtrees from (S_i, S_j)(line 15-17) and add numerical weight, $\delta > 1.0$, one by one. In terms of the evolution of the weight adjustment, each dimension in the feature vectors represents a specific type of AST nodes and is the occurrences 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 δ for that specific location correspondingly (line 18-22). We initialize the weight δ as a random number which is greater than 1.0 and re-calculate the euclidean distance between two feature vectors. We repeat this process until the distance is out of current code similarity threshold S (line 19-20). This is designed to guarantee that these two code samples will not be considered as code clones 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.

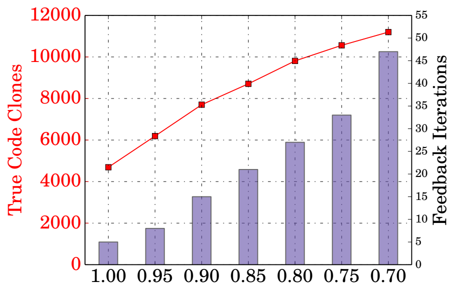
Example: Here, we give an example to illustrate how our formal feedback works. We reuse the false positive example showing in Figure 5.4. As we have described in Section 5.1.1, these two trees share a common tree pattern but with 2 different nodes (showing in shady color) out of 17 total nodes. Assuming 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, these two code fragments will be identified as clones if $S = 0.75$. During the feedback loop, we first identify these 2 different nodes in each tree by finding the LCS. Assuming we initial the weight $\delta = 2$ and add it to the corresponding dimension in the feature vectors, we can obtain the updated feature vectors as $< 7, 1 + 1 \times \delta, 1 + 1 \times \delta, 1 + 1 \times \delta, 0, 1, 1, 1, 1 >$ and $< 7 + 1 \times \delta, 1, 1, 2, 1 \times \delta, 1, 1, 1, 1 >$. We then re-calculate the euclidean distance of these two updated feature vectors, and they will be no longer satisfied within the threshold $\sqrt{2(1 - S) \times min(S(C_i), S(C_j))}$. Thus, we can eliminate such false positives in the future.

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.

5.3 Evaluation



(a) thttpd



(b) links

Figure 5.6: 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

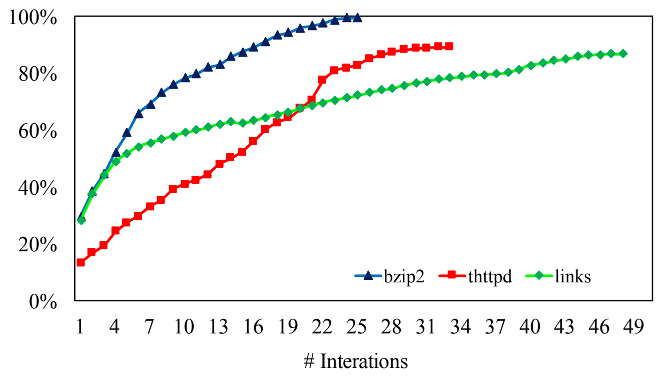


Figure 5.7: Accumulated percentage of false positives eliminated by Twin-Finder with code similarity set to 0.70

This section presents a detailed evaluation results of Twin-Finder against a tree-based code clone detection tool DECKARD [55] in terms of code clone detection, and conduct several case studies for applications security analysis.

5.3.1 Experiment Setup

We performed empirical experiments on Twin-Finder. We selected 7 different benchmarks from real-world applications: bzip2, hmmer and sphinx3 from SPEC2006 benchmark suite [1]; man and gzip from Bugbench [75]; thttpd-2.23beat1 [4], a well-known lightweight sever and a lightweight browser links-2.14 [95]. 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. [55], setting minimum token number (minT) as 20, stride to infinite, and code similarity is set between 0.70 and 1.0.

5.3.2 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 5.2: 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 5.3: 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.

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 5.4: Statistics of code clones detected from Twin-Finder with the number of iterations for feedback until converge where S is the code similarity

We evaluated the effectiveness of Twin-Finder to show the optimal results Twin-Finder is able to achieve. The code similarity is set as 0.80 with feedback enabled to eliminate false positives until converge (no more false positives can be observed or eliminated) in the first experiment. Table 5.2 shows the size of the corresponding percentage of more code clones detected using our approach. As we can see, the results show that Twin-Finder is able to detect 393.68% more code clones on average compared to the clone detection without slicing and feedback, with the lowest as 128.80% in *thttpd* and highest up to 913.89% in *gzip*. Note that our approach achieves the best 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 5.3. 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 5.6 presents the number of true code clones detected in thttpd and links from our approach (drawn as the red line in each figure) and the number of iterations for feedback needed to converge (shown as the bar plot in each figure) correspondingly. We also repeated the same experiments with three different code similarities setups in other smaller benchmarks. Table 5.4 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.

5.3.3 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 5.7 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 to different programs due to different program behaviors and program size. As the results show, more feedback iterations are needed for larger programs 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 across 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.

5.3.4 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 5.5 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.

5.4 Summary

In this chapter, 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 Type	**Source File**	**Function Name**	**Pointer Name**	**Bug Report**	**Exploitation**
Null Dereference	Links/language.c	get_language_from_lang	lang	Not Reported	All three cases use memory allocation functions, which can be return NULL to indicate an error status.
Null Dereference	Links/language.c	get_language_from_lang	p	Not Reported	
Null Dereference	Links/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/ftp.c	ftp_logged	rb	Not Reported	All those three functions use dynamic allocations, however never being freed after.
Memory Leak	Links/bfu.c	do_tab_compl	items->text	Silently patched	
Memory Leak	Links/terminal.c	add_empty_window	ewd	Not Reported	
Buffer Overflow	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 5.5: Using Twin-Finder to test Links-1.4 and libreoffice-6.0.0.1

bug in libreOffice-6.0.0.1.

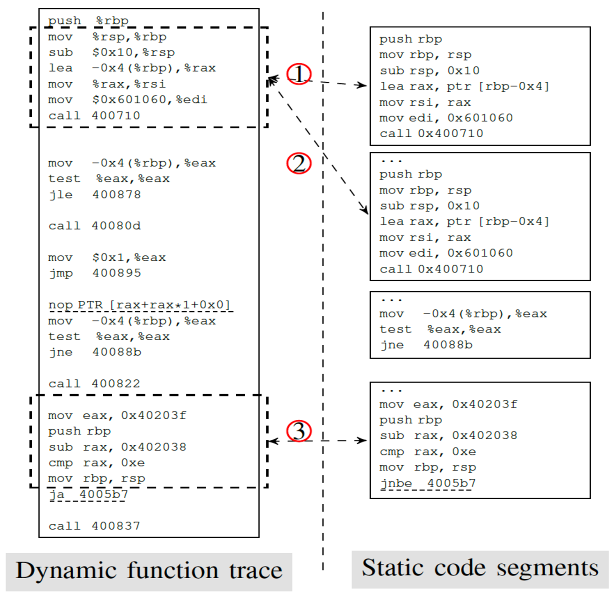
Chapter 6 Integrated Deep Learning Approach for Attack Surface Reduction in Program Binaries

In this chapter, we propose Hecate, a framework that leverages dynamic execution and trace to create customized, self-contained programs to minimize the corresponding attack surface. A key feature of Hecate is that it makes novel use of deep learning to identify program and communication-related features in binary in an automated fashion. It employs the test-cases to invoke different program features, applies trace splicing to extract dynamic execution paths (of invoked features) from the complete instruction trace, maps the paths to owner functions in the binary code, finally identifies program features (as targets for customization) through their constituent functions.

6.1 Background

Software customization comprises two tasks: (i) identifying program features from a binary executable by analyzing and mapping dynamic instruction trace that invokes different features, and (ii) tailoring and rewriting the binary, in accordance with user needs, to create customized, self-contained programs.

The goal of Hecate's feature identification is to map dynamic instruction trace (relating to different features) to feature-constituent functions in binary. Ideally, it is possible to log the virtual addresses of each executed instruction. Then we can get the memory layout of each binary module (e.g., through /proc/pid/ma ps on Linux). With these two pieces of information, we could uniquely map a dynamic trace back to static code. However, there are some scenarios in practice where the address is not available. For example, commercial software and operating system are usually slightly obfuscated to deter reverse engineering and unlicensed use. Further, system and kernel libraries are often optimized to reduce disk space requirements[48, 106]. It may be difficult to even locate function entry points (FEPs) since the full symbol or debug information is usually not available in optimized binaries [13]. Thus, we have to


```
push  %rbp
mov   %rsp,%rbp
sub   $0x10,%rsp
lea   -0x4(%rbp),%rax
mov   %rax,%rsi
mov   $0x601060,%edi
call  400710


mov   -0x4(%rbp),%eax
test   %eax,%eax
jle   400878

call  40080d

mov   $0x1,%eax
jmp   400895

nop PTR [rax+rax*1+0x0]
mov   -0x4(%rbp),%eax
test   %eax,%eax
jne   40088b

call  400822

mov  eax, 0x40203f
push rbp
sub  rax, 0x402038
cmp  rax, 0xe
mov  rbp, rsp
ja   4005b7

call  400837
```

```
push rbp
mov rbp, rsp
sub rsp, 0x10
lea rax, ptr [rbp-0x4]
mov rsi, rax
mov edi, 0x601060
call 0x400710
```

```
...
push rbp
mov rbp, rsp
sub rsp, 0x10
lea rax, ptr [rbp-0x4]
mov rsi, rax
mov edi, 0x601060
call 0x400710
```

```
...
mov   -0x4(%rbp),%eax
test   %eax,%eax
jne   40088b
```

```
...
mov  eax, 0x40203f
push rbp
sub  rax, 0x402038
cmp  rax, 0xe
mov  rbp, rsp
jnbe  4005b7
```

Figure 6.1: An illustrative example of feature identification by mapping dynamic instruction trace to functions in static code from OpenSSL.

utilize code patterns to match dynamic traces. This is a challenging problem because dynamic trace and static code often have different patterns and cannot be accurately matched through techniques such as execution path alignment [79]. Consider the example shown in Figure 6.1 with dynamic instruction trace and binary code snippet from OpenSSL. First, as Arrows 1 and 2 indicate, the same basic block from dynamic instruction trace could have multiple matches in the binary, and cannot be uniquely mapped to a single function. Second, the same binary instruction can be interpreted into different verbal presentations, in which case different disassemblers will give different outputs. As Arrow 3 indicates, the binary value 77H can be translated to the opcode either “ja” (jump above) or “jnbe” (jump not below), causing direct pattern matching to fail. Further, when loops and recursive function calls exist in the binary, it is difficult to correctly identify these structures in dynamic instruction trace. We conducted an experiment using a substring matching approach to map the opcode pattern between instruction traces and binary code. Examining two applications, bzip2 and OpenSSL, function mapping techbniques only achieves an average accuracy of 76.31% and 73.02%, respectively.

6.1.1 Problem Statement

To introduce our problem of software customization, we first need a definition of what a feature is in binary code.

Definition 8. Function. The term *function* in this paper particularly refers to the function identified in static binary code, which is a collection of basic blocks with one entry point (i.e., the next instruction after a call instruction) and possibly multiple exit points (i.e., a return or interrupt instruction). All code reachable from the entry point before reaching any exit point constitutes the body of the assembly function. For a given program, we use $\mathcal{F} = \{f_k,\ \forall k\}$ to denote the set of all functions existing in the static binary code.

Definition 9. Feature. A program feature is defined as a set of constituent functions – denoted by $F_i = \{f_i^1, f_i^2, ..., f_i^n\} \subseteq \mathcal{F}$ – which uniquely represent an independent, well-contained operation, utility, or capability of the program. A feature at the binary level may not always correspond to a software module at the source level. We use $\mathcal{T} = \{F_i,\ \forall i\}$ to denote the set of all available features in the program.

The goal of Hecate is that, given a program binary, test cases invoking program features, and user's customization requirement (i.e., a set of desire features $\hat{\mathcal{T}} \subseteq \mathcal{T}$), it will produce a modified binary that contains the minimum set of functions to satisfy the user's requirement and to support all desired features in $\hat{\mathcal{T}}$. We perform the customization after abstracting the program into Control Flow Graph (CFG). From the perspective of CFG, the customized binary is composed of a CFG that is a subgraph of the original program CFG.

6.2 Hecate System Design

6.2.1 Approach and System Architecture

Hecate consists of two major modules: feature identification and feature tailoring. Its system architecture is illustrated in Figure 6.2. Users provide their requirements

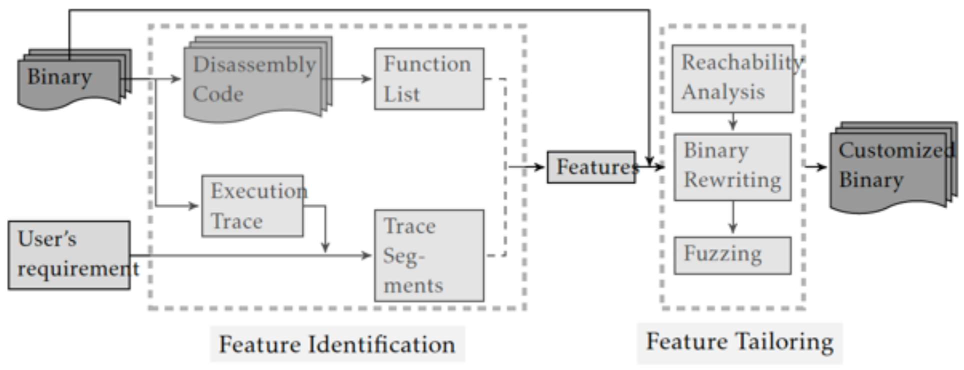


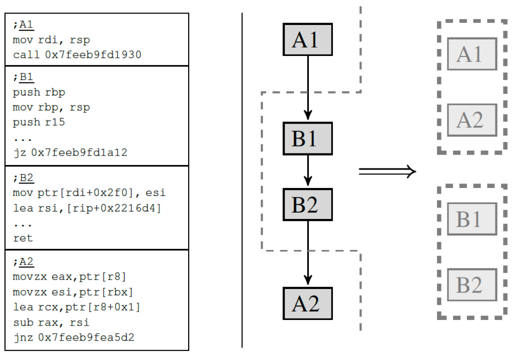
Figure 6.2: Hecate System Diagram

(i.e., a list of features that are needed) as well as test-cases to reach different features. Hecate takes the program binary and customization requirement as inputs and generate a customized binary consisting of only the desired features. For feature identification, Hecate first builds a function library based on static analysis of program binary, including dynamically linked libraries. Byteweight, a learning-based binary analysis tool, is employed to identify function body directly from static program binaries. Next, execute the program using the test-cases provided, analyze the dynamic instruction trace, extract execution paths relating to different features (or feature combinations), and maps them to constituent-functions in the program binary.

The feature tailoring module is explained in section 6.2.3. It modularizes program features through their constituent functions and modifies the program binary in accordance with user's customization requirements. The CFG of the customized program can be viewed as a sub-graph of that of the original program, which is able to retain the behavior of only the desired features. At last, a fuzzing engine can be employed to generate inputs and further test the customized binary.

6.2.2 Feature Identification

Feature Identification uses trace splicing to extract dynamic execution paths and maps them to owner functions in the binary code, enabling us to identify program features through their constituent functions. In this paper, we define an *execution path* as a sequence of instructions that are executed from a function entry point to



```
;A1
mov rdi, rsp
call 0x7feeb9fd1930
;B1
push rbp
mov rbp, rsp
push r15
...
jz 0x7feeb9fd1a12
;B2
mov ptr[rdi+0x2f0], esi
lea rsi,[rip+0x2216d4]
...
ret
;A2
movzx eax,ptr[r8]
movzx esi,ptr[rbx]
lea rcx,ptr[r8+0x1]
sub rax, rsi
jnz 0x7feeb9fea5d2
```

Figure 6.3: Extracting dynamic execution paths of each individual function through trace splicing. Boxes stand for basic blocks. A_1 and A_2 belong to function A while B_1 and B_2 belong to function B.

an exit point. The function containing the execution path is known as the *owner function*. Our approach leverages deep learning and works in a fully unsupervised, autonomous fashion.

6.2.2.1 Function Recognition

We first construct the pre-image and image of our function mapping, using trace splicing and deep-learning tools, respectively. The pre-image is defined as the set of execution paths obtained from dynamic instruction trace, while the image is defined as the set of functions recognized in static program binaries.

We run the target executable with provided test cases to invoke different (combinations of) program features, and collect instruction trace to capture the dynamic execution of the program. The trace is then spliced to extract execution paths belonging to different functions, which serves as the pre-image of our function mapping. Consider the illustrative example shown in Figure 6.3, where a sequence of 4 basic blocks, A_1, B_1, B_2, A_2, are captured in dynamic trace, when function f_B is called inside function f_A. Clearly, we cannot directly map the entire sequence to functions in binary code, because it contains two separate execution path, belonging to functions f_A and f_B, respectively. We employ two different methods to splice dynamic trace and extract different execution paths: (1) We track call stack changes together with instruction trace. By recognizing *push* and *pop* operations on the call stack, we can infer function call events, and slice and associate basic blocks that belong to the same function. (2) From the instruction trace, instructions that perform function calls and returns will be recognized and put embedded function calls into different layers.

We remove duplicate basic blocks in execution traces to improve the accuracy of function mapping. Furthermore, every time a function is invoked, a different execution path may be traversed inside the function. These execution paths will be separated and mapped to their owner functions independently, minimizing the probability of false negative in function mapping. In Hecate, we unitize ByteWeight [13], a learning-based tool that identifies function bodies from binary.

6.2.2.2 Function Mapping

In Hecate, we leverage deep learning approach to propose a solution to enable automated function mapping. We use the approach, mentioned in 4.2.2, to model binary instruction sequences using Recursive Neural Network (RNN). The framework is constructed with two key components. First, to obtain vector embedding for a given execution path (that consists of an instruction sequence), we use RNN to map each term in the binary instructions (e.g., opcodes and operands) to a vector embedding at the lexical level, resulting in a signature vector for the entire execution path. Second, we consider the mapping problem as a multi-class classification problem, where each function is considered as a class label, different execution paths obtained from the function's binary code as samples of that class, and an execution path extracted from dynamic instruction trace as the testing sample. We employ a multi-class Convolutional Neural Network (CNN) classifier to identify the owner functions of an arbitrary dynamic instruction trace. Our deep learning approach is inspired by the related work on source code analysis [117, 107, 110, 41]

For a given execution path with multiple terms and instructions, we adopt a greedy method [101] 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 instruction, and then combines the results from a sequence of instructions in an execution path.

Multi-class classification for function mapping. Function mapping aims to recognize the owner function (in static binary) of a given execution path obtained from the dynamic trace. We consider each function as a class label, different execution paths obtained from the function binary code as samples of that class, and an execution path extracted from dynamic instruction trace as the testing sample. Then, the mapping becomes a multi-class classification problem, which is solved using Convolutional Neural Networks (CNN) in this paper. We adopt the sentence classification model proposed in [60] for natural language processing and train a multi-class classifier using CNN for function mapping. Note that another line of work, such as tainting [118, 96, 116], can be used for feature identification. We consider this as future work.

To obtain training samples for each class, we use CFG analysis to construct different execution paths for each function identified in the binary code. More precisely, once the function boundaries and bodies are recognized, we use a Depth First Search (DFS) to traverse the static CFG of each function and construct related execution path using a *random walk*.

6.2.3 Feature Tailoring

Feature tailoring creates customized software that consists of the desired features and their constituent functions in accordance with user needs. It has to address a number of challenges. First, a single execution trace may not reach all desired features, requiring us to merge multiple outputs from feature identification. Second, different features often share some common constituent functions. If the goal of tailoring is to remove certain features, we need to identify and retain the shared functions in the customized binary.

6.2.3.1 Methods for Feature Tailoring

Let $\hat{\mathcal{F}}$ be a set of target program features for tailoring. If the constituent functions of each feature $F_i \in \hat{\mathcal{F}}$ can be successfully identified, we can simply create a superset of their constituent functions, i.e., $\hat{F} = \cup F_i$. Two techniques are developed next to

(i) create a customized program by retaining only the features in $\hat{F}$ (e.g., if user only needs these features) and (ii) remove the features in $\hat{F}$ from the binary (e.g., if they are deemed as unnecessary or vulnerable). When $\hat{F}$ cannot be directly identified, we leverage set operations, including union, intersection, and subtraction, to construct $\hat{F}$ from available feature combinations, in order to fulfill feature tailoring.

Retaining features. We consider the case where a user only needs a set of features $\hat{\mathcal{F}}$. To deliver the tailored program, we execute the original program with available test-cases to generate dynamic traces that reach each feature in $\hat{\mathcal{F}}$. Through feature identification, we identify the set of constituent functions F_i of each feature i and derive the superset $\hat{F} = \cup F_i$, which is the set of functions we need to retain in the customized binary. Due to possible missing constituent functions during feature identification and deep learning, the set $\hat{F}$ may not contain all necessary functions to execute the resulting binary. We apply static CFG analysis to find an add any required functions that makes $\hat{F}$ complete. In particular, each function in $\hat{F}$ will be mapped to the pre-built static CFG and the reachability analysis in section 6.2.3.2 will ensure that each mapped node in the CFG can be reachable from the function start.

Removing features. We consider now removing a set of features $\hat{\mathcal{F}}$ from a given program binary, which is useful when a user deems these features either unnecessary or vulnerable. To this end, we again execute the binary with test-cases to reach each unwanted feature in $\hat{\mathcal{F}}$. Then, after identifying each F_i from dynamic trace, the superset of constituent functions $\hat{F} = \cup_i F_i$ that correspond to the unwanted features can be obtained. However, for feature removal, we cannot simply eliminate all functions in $\hat{F}$ from the binary, due to the existence of shared functions with other (desired) features, which are required for the soundness of the customized program. Let $\hat{S}$ be the set of functions/basic blocks shared by other features (which can be found using the constituent functions of other features). Hecate will only remove functions/basic blocks in $\hat{F} - \hat{S}$, which are only needed for the operation of the unwanted features.

Tailoring via set operations. When the target features' constituent functions $\hat{F}$ are not directly identifiable, Hecate employs set operations including union, intersection, and subtraction to compute $\hat{F}$ from known feature combinations. **Union:** A feature may contain multiple execution paths that cannot be dumped and identified in a single execution. Hecate will collect traces from different program executions to identify and compute the union of the related feature-constituent functions. **Intersection:** A program may contain concurrent features that cannot be identified separately from the available execution trace. For instance, OpenSSL's *choosing cipher suite* feature is always coupled with the execution of encryption/hash functions in dynamic trace. To identify the constituent functions of *choosing cipher suite* feature, we can take the intersection of multiple executions with different choices of encryption/hash functions. **Subtraction:** This operation allows us to identify the unique constituent functions of given features. So, we can safely remove them without affecting the soundness of other features due to shared functions.

6.2.3.2 Reachability Analysis

Algorithm 3 Reachability Analysis

```
Static CFG: $\mathscr{G} = \{\mathscr{V}, \mathscr{E}\}$
Initial feature set:$\mathscr{F} = \{F_0, F_1, ..., F_n\}$
$\mathscr{F}'$=final feature set
Initialization:$\mathscr{F}' = \mathscr{F}$
for $F_k$ in $\mathscr{F}$ do
    Find $V^f$: $V^f \supset F_k$ && $V^f \in \mathscr{V}$
    Find $\mathscr{T} = \{T_{head}, T_1, ..., T_m, T_{tail}\}$: $\mathscr{T}$ is the control flow path that contains $F_k$
    if ($V^f.entry \geq 2$ || $V^f.exit \geq 2$) && $\mathscr{F}$ is for feature removal then
        $\mathscr{F}' = \mathscr{F}' - \{V^f\}$
    if $\exists\mathscr{T}$ then
        $\mathscr{F}' = \mathscr{F}' \cup \mathscr{T}$
```

A program's CFG can be represented as a directed graph, $\mathscr{G} = \{\mathscr{V}, \mathscr{E}\}$, where the node set $\mathscr{V} = \{v_1, v_2, ..., v_m\}$ represents basic blocks and edge set $\mathscr{E} = \{e_1, e_2, ..., e_n\}$ represents control flows among basic blocks. The customized program can be viewed as a subgraph $\mathscr{G} = \{\mathscr{V}, \mathscr{E}\}$, for $\mathscr{V} \subseteq \mathscr{V}$ and $\mathscr{E} \subseteq \mathscr{E}$. Ideally, for a given set of desired features, Hecate's feature identification and tailoring modules should obtain

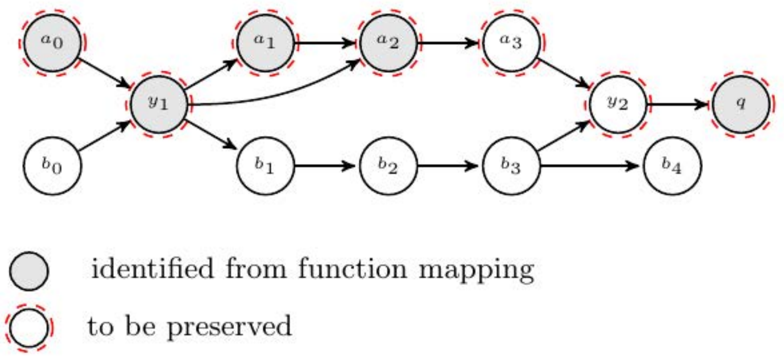


Figure 6.4: Reachability analysis on LibreOffice: retaining feature

their feature-constituent functions $\hat{F}$ that meet the following two requirements: (i) All functions in $\hat{F}$ should belong to desired features; (ii) The functions in $\hat{F}$ together can ensure that the desired features are functional, i.e., the customized binary can be executed with inputs that can reach the desired features. However, these may always hold because deep-learning-based algorithm cannot guarantee to always produce the correct function mapping and feature identification. Necessary functions for the soundness of the customized binary may be missing, causing the program to crash and unable to execute the desired program features. We propose a CFG-based reachability analysis to tackle the issue. We design an algorithm as shown in Algorithm 3 to rectify possible missing functions and ensure the soundness of customized binary by expanding the identified feature functions. The basic idea is to connect the missing links in the CFG and preserve the shared code segments. As Algorithm 3 shows, we apply the following methods in the CFG: (i) If the basic block can jump to multiple targets ($V^f.exit \geq 2$), or multiple basic blocks can jump to this basic block ($V^f.entry \geq 2$), then this basic block is considered as a shared code segment. Hence, this basic block will not be removed in any case.

In the example shown in figure 6.4 from LibreOffice, where two features are intertwined. The circles represent the basic blocks and gray circles are those identified by feature identification module. The feature $\mathscr{F} = \{a_0, y_1, a_1, a_2, q\}$ is the desired feature that the user wants to keep. Without reachability analysis, a_3 and y_2 won't be kept in the customized binary since they are not identified by deep learning mapping. According to Algorithm 3, the identified basic blocks that belong to the same function will be connected by adding the missing nodes along the control flow. We define $\mathscr{T}$ as the control flow path that resides within the scope of one function and contains all elements in $\mathscr{F}$. In this case, all the basic blocks in $\mathscr{T} = \{y_1, a_1, a_2, a_3, y_2\}$ should be included in $\mathscr{F}'$ even if a_3 and y_2 are not discovered by feature identification module. The nodes with red dashed circles are the final elements in the feature set to be customized.

6.2.3.3 Binary Rewriting

We use feature tailoring to derive a set of functions to eliminate in program binary. Simply replacing these function bodies with "NOP"s would not generate a valid executable, because (i) some code segments in the eliminated function body may be shared with other functions, and (ii) there may exist data segments that are inserted into the eliminated functions and must be preserved.

To address these issues, Hecate utilizes a static binary rewriter, DynInst, to modify the program binary by rewriting the binaries in basic blocks level in the CFG. As DynInst is capable to abstract the program basic blocks in the form of CFG. To remove the features in the programs, there are two steps in Hecate. First, Hecate removes the functions that should not be called. The call site of the eliminated functions will be replaced to redirect the program to exit point. Second, for those functions cannot be removed from the first step (e.g., For indirect function calls, the address of the callee function cannot be decided beforehand and can potentially lead to any other addresses), we replace the rest of the function body with "NOP". Furthermore, a verification process is performed using program fuzzing approaches [125] by Hecate to validate the effectiveness and correctness of feature tailoring. Specifically, the fuzzing engine generates two sets of test cases: (1) F_1 that invoke the desired features in customized program; (2) F_2 that involve at least one of the eliminated features. In particular, Hecate usesF_1 to confirm the integrity of necessary program functionalities, while F_2 helps verify the successful removal and handling of eliminated features.

6.3 Evaluation

6.3.1 Experiment Setup

Our experiments are conducted on a 2.80 GHz Intel Xeon(R) CPU E5-2680 20-core server with 16 GByte of main memory. The operating system is Ubuntu 14.04 LTS.

Benchmarks. In our evaluation, we select three sets of real-world applications: (i) Non-interactive applications including two applications from SPEC 2006 Benchmark suite [1], bzip2 and hmmer; two applications from a bug benchmark suite *bug-bench* [75], polymorph and man and (ii) Interactive applications including a light-weight web server thttpd, version beta 2.23, an open source office suite LibreOffice and a web browser links. (iii) An implementation of Transport Layer Security (TLS) & Secure Sockets Layer (SSL) protocol, OpenSSL.

Dataset and Training. In our function mapping module, we collect static execution paths as training dataset and dynamic execution paths as testing dataset for evaluating the accuracy of the pre-trained models. We selected the highest quality model and extracted the matrix of embeddings. We have observed that a well trained function mapping model is with the hidden node size as 500 in RNN and 200 maximum iterations for RAE, which is chosen as the parameters of deep neural network in function mapping module.

6.3.2 Accuracy of function mapping

In this section, we evaluate the accuracy of the pre-trained function mapping module in Hecate and presents the accuracy of five representative applications. We construct the testing dataset as follows: We collect the dynamic instruction traces for each identified function in the binary and perform the same *random walk* process to generate execution paths as mentioned in Section 6.2.2.2. Table 6.1 shows the statistics for our subject benchmarks. The testing dataset size is controlled to be 30% as big as the training dataset reported in Table 6.1. We also observed that due to the different amount of training data we can obtain from different functions, the

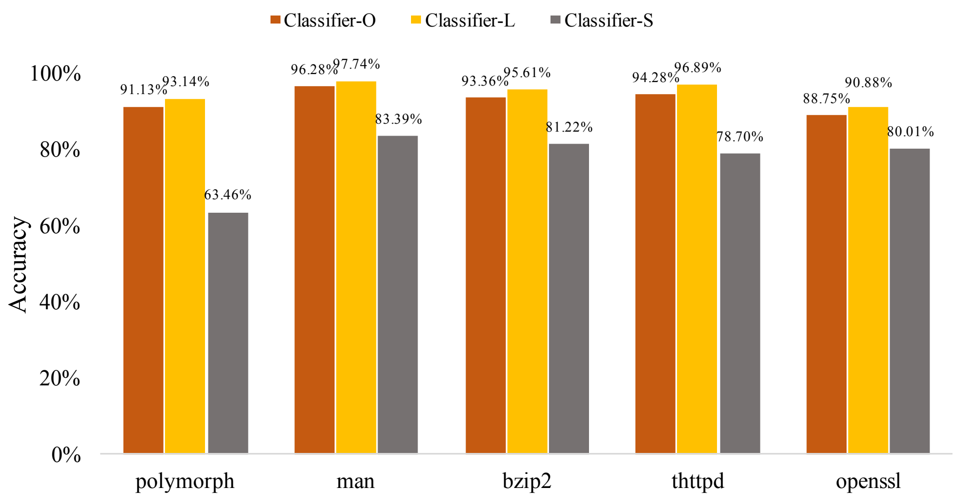


Figure 6.5: Accuracy of function mapping during feature identification

mapping accuracy will be higher if we split functions into large and small categories, by using the median number of training data sample size. We trained three CNN classifiers for each application, one is trained cross all the functions as an overall classifier (Classifier-O), and the other two are trained for large functions (Classifier-L) and small functions (Classifier-S) respectively.

The function mapping accuracy is plotted in Figure 6.5. We achieve an overall average accuracy of 92.76%, with the highest up to 96.28% in *man* from *bugbench*. In general, the mapping accuracy of larger programs, such as bzip2 and thttpd, is higher than smaller programs like polymorph. Because the number of execution traces used for training our CNN classifiers in those programs is much larger than that in polymorph, there are 189,855 training execution paths in bzip2 comparing to 10,806 in polymorph). For the applications with more functions, such as OpenSSL that has 4,023 functions, the overall accuracy can be as low as 88.75% since there are more classes for classification. We also note that all of the Classifier-Ls outperforms the Classifier-Os. For instance, in *polymorph*, the accuracy of Classifier-L is 93.14% whereas the accuracy of Classifier-O is 91.13%. However, we observe that the accuracy for Classifier-S is lower than Classifier-L. The reason is that functions trained in Classifier-Ss are relatively small, with limited training data samples for classification. In particular, the accuracy of Classifier-S is 63.46% for polymorph, which is the worst among all the applications. We further analyzed and found that the median number

Benchmark	**#Functions**	**Vocabulary Size**	**#Training Execution paths**	**#Tokens**
polymorph	23	201	10,806	460,248
man	77	1,198	346,570	86,008,653
bzip2	79	1,251	135,738	54,809,155
thttpd	129	1,838	189,855	54,162,084
OpenSSL	4,023	10,582	586,817	137,293,197

Table 6.1: Benchmark Profiles

Bench.	*polymorph*	*bzip2*	*hmmer*	*thttpd*	*links*	*LibreOffice*	*OpenSSL*
Program Size (LoC)	404	5,904	20,721	7,956	178,441	4,485,797	305,279
#Selected Features	6	8	7	12	12	10	14
#Combinations	64	96	48	3,072	6,144	768	4,096

Table 6.2: Number of identified features and customized programs by Hecate.

of training data size is 7 for polymorph, which means almost half of the functions have only less than 7 training data samples. The lack of training data leads to a bad performance for classification.

6.3.3 Feature Combinations

In this section, we evaluate the number of customized programs after feature tailoring. When the features are identified by customers, we can create multiple customized binaries containing different feature combinations. Table 6.2 shows the number of selected features for each benchmark and the number of customized programs we are able to create. Our approach is able to produce numbers of customized programs to match the customer needs while minimizing unwanted exploitation of the applications features. The number of customized programs is calculated after feature tailoring. Since each feature can contain both unique functions and shared functions as mentioned in Section 6.1, there are some scenarios that several features cannot be customized separately. For example, the 6 features we selected in polymorph are all independent to each other and totally separable. Hence we are able to create 2^6 customized programs. However, in LibreOffice, there are 2 selected features: print files and print files to a specific printer, they both execute print feature and have shared

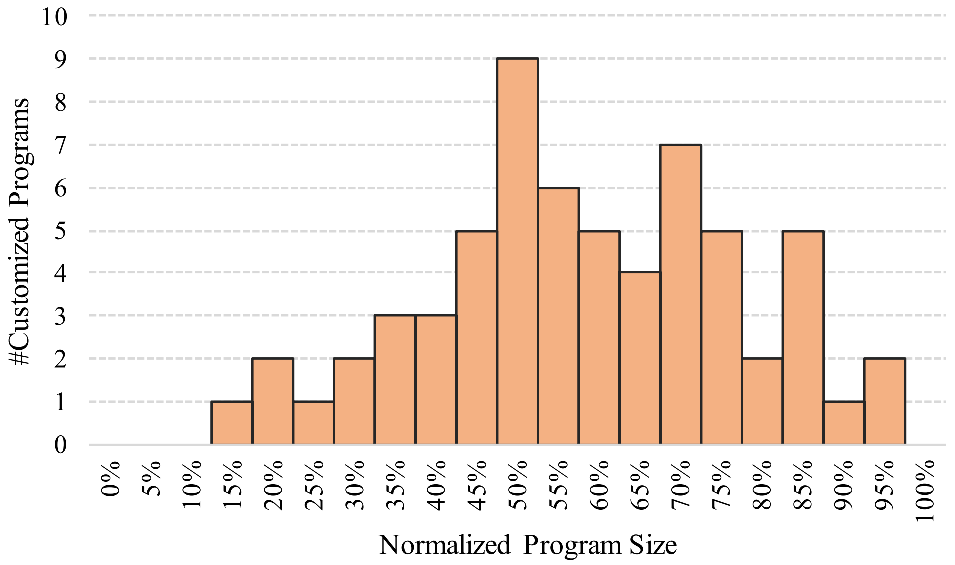


Figure 6.6: Number of customized program versions and their sizes normalized to original program (polymorph benchmark)

functions. They either can be removed or retained together. Thus, Hecate cannot create a customized version of LibreOffice with arbitrarily feature combinations.

We also evaluate the size of customized program variations. We pick one example program polymorph(a Win32 to Unix filename converter) to present the result. We identified 6 features in polymorph as: convert file, convert all hidden files, clean files, help/usage, trace file path, and program version. Figure 6.6 shows the program size distribution in terms of normalized program size in polymorph. As we can see, we generate various combinations of customized programs that contain *just-enough* software features to support specific use-cases and can significantly reduce the program size up to 85%.

6.3.4 Impact on program security

We evaluate the impact of feature customization on program security here. As shown previously, the reduction of code size also shrink the attack surface and eliminate possible vulnerabilities in programs. We survey the known CVEs of different programs that can be removed by feature customization. For instance, in OpenSSL, i) the *CVE-2014-0160*, known as *Heartbleed* bug, can be eliminated by removing the *heartbeat* extension; ii) the *CVE-2016-7054*, which can lead to DoS attack can be

Program	**# Removed CVEs**	**% Features removed**
OpenSSL(2014-2017)	45	44.6
LibreOffice	23	67.6
Thttpd	5	38.5
Bzip2	2	22.2

Table 6.3: Impact on Application and Communication security

neutralized by removing **-CHACHA20-POLY1305* ciphersuites; iii) the *CVE-2016-0701*, which can cause information leakage, can be negated by avoiding using DH ciphersuites; The *CVE-2015-5212* in LibreOffice (an integer underflow bug) can be removed by disabling the printer functionality when users don't need it.

In total. we found 101 CVEs in OpenSSL distributions during 2014-2017, 34 CVEs in LibreOffice, 13 CVEs in Thttpd and 9 CVEs in Bzip2. Not all vulnerabilities can be disabled by our feature customization. Some vulnerabilities are in the functions that are necessary for program execution. *CVE-2010-0405* in Bzip2 is an integer overflow bug in function *BZ2_decompress*. In most of the cases, decompression is a feature that users will not remove. The number and ratio of program features that can be removed are shown in Table 6.3. We evaluate the security impact of Hecate using the ratio of CVEs that can be removed by feature customization.

6.4 Summary

We design and evaluate a binary customization framework Hecate that aims to generate customized program binaries with *just-enough* features and can satisfy a broad array of customization demands. Feature identification and feature tailoring are two major modules in Hecate, with the former one discovering the target features using both static code and execution traces, and the latter one modifying the features to reconstruct a customized program. Our experiment results demonstrate that Hecate is able to identify features with the highest accuracy up to 96.28% and reduce the attack surface by up to 67%.

Chapter 7 Related Work

7.1 Redundant bound checks elimination

A number of related work including Softbound [80], ABCD [18] and WPBound [92] have been discussed in previous sections. Besides these, static code analysis has been widely used for program vulnerabilities discovery, with different approaches and tools proposed in *Chucky* [113], *Splint* [37] and *PScan* [7]. Nurit et al. [35] targeted string-manipulation related bugs in C program with a conservative pointer analysis similar to ABCD, using an abstract constraints expressions for pointer operations. Such static approaches require extensive program modeling and analysis (e.g., by constructing constraint systems) and also fall short on dealing with certain vulnerabilities that occur only at runtime (e.g., due to user input-related bugs). Another line of work employs machine learning to improve the efficiency of static code analysis. In particular, Fabian et al. [113] [112] used machine learning to identify the similarity of code patterns to facilitate vulnerable discovery. In these approaches, the accuracy for resulting vulnerabilities discovery relies on the choice of machine learning algorithms, which often cannot guarantee zero false positives. The introduction of alternative code patterns for infrequently used code can help to reduce false postive [93].

To the best of our knowledge, most of the existing approaches that have been proposed for bounds violations detection or other bounds-related vulnerabilities are pure static code analysis, such as [44, 31, 6, 73, 72, 8, 82, 39, 115, 67]. More precisely, Thomas et al. [103] used dominator tree to maintain the conditions based on code blocks. But different from ABCD, they do not construct inequality graph and only keep a condition for every instrumented instruction for each code block. George et al. proposed CCured [83], which is a type safe system that leverages rule-based type inference to determine "safe" pointers. It classifies pointers to three types $\{safe, seq, dynamic\}$ then applies different checking rules for them. Again, these works are relying only on static code analysis, while none of them capitalizes on runtime information for efficient redundant condition inference and bounds check elimination.

7.2 Code Clone Detection

Code clone detection techniques can generally be classified into several categories. String matching-based techniques [36, 11, 12, 65] 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 [62, 100, 99, 15, 123, 66] 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 [58], DECKARD [55] and CP-Miner [71]. Learning based approaches have also been developed for code similarity detection. White et al. [101] proposed a deep neural network (DNN) based code clone detection in source code. Komondoor et al. [61] 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 [105] 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.

7.3 Code analysis and De-bloating:

Several prior works have proposed program customization frameworks only based one method like de-bloating [56], cross-host tainting [29] and so on. In terms of binary reuse, it has been studied by several works [109]. The main challenge of reusing binary code is it only focuses on reusing partial code in the program's high-level assembly code. Some existing works try to find memory-related vulnerabilities in source code or IR by direct static analysis [114, 97, 98, 21, 51, 28, 70, 119]. BinRec [63] reconstructs intermediate representation (IR) of program binaries to allow complex transformations in code and reduce attack surfaces. While our proposed approach improves the efficiency of customizing binaries through the use of machine learning, BinRec improves effectiveness using compiler-based tools. As such, the two approaches are quite complementary and when combined together, can present an improved framework for eliminating attack surfaces in programs.

7.4 Deep Learning and Language Modeling

The state-of-the-art Deep Learning algorithms have been used as new approaches for language modeling [57, 40, 69, 5, 64]. 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 [42, 5, 104, 50, 122, 114]. For instance, recurrent neural network (RNN) is known as a capable approach for modeling sequential information [47, 89, 53, 54, 74, 38, 34]. Recently, such techniques have been applied on modeling program source code fragments. White et al. [101] 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.

Chapter 8 Conclusion and Future Works

With the rapid growth of software systems, securing such applications has become very challenging due to the growing software complexity. Two traditional lines of code analysis techniques that have some fundamental limitations. Pure statistical methods rely on probabilistic inference and often fail to guarantee complete accuracy. Formal methods require exhaustive analysis along all paths in the application code, which can be prohibitively expensive in terms of time and resources.

This dissertation proposes Learn2Reason, a novel joint learning approach, an integration of statistical and formal methods without the unification of the underlying knowledge representation. The main contributions of this dissertation are to improve the security issues of code analysis by integrating statistical analysis and formal methods, thereby reducing the time-to-solution.

We first proposed two novel techniques that aim to secure memory safety in both source code and binary executables. The SIMBER framework integrates with statistics-guided inference with spatial memory safety checks to perform a function-level redundant bounds check removal during runtime. Its statistical inference adaptively builds a knowledge base (i.e., a safe region for eliminating redundant bounds checks). Currently, SIMBER works at function-level granularity. For future work, it can be applied to finer granularity bound checks removal. SIMBER achieves a significant redundant bound checks removal rate and guarantees there are no false positives. We evaluated SPEC benchmarks with high softbound overhead and their Check-Hotspot functions. Finally, SIMBER obtained an average 76.5% of dynamic bound checks removal rate and 65.31% execution time reduction. Clone-Hunter technique integrates a machine learning-based binary code clone detection to speedup elimination of redundant array bound checks in binary executables. We evaluated our approach using real-world applications from SPEC 2006 benchmark suite. Our results show the time-to-solution (the time spent to remove bound checks) for Clone-Hunter is 90× faster compared to pure Binary Symbolic Execution while three out of four applications fail to finish the execution.

Second, we discovered a new type of code clone detection framework Clone-Slicer that identifies domain specific code clones. 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. On the other hand, we introduced a pointer-related code clone detector, Twin-Finder, 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.

Additionally, we design and evaluate a binary customization framework Hecate, that aims to generate customized program binaries with *just-enough* features and can satisfy a broad array of customization demands. Feature identification and feature tailoring are two major modules in Hecate, with the former one discovering the target features using both static code and execution traces, and the latter one modifying the features to reconstruct a customized program. Our experiment results demonstrate that Hecate is able to identify features with the highest accuracy up to 96.28% and reduce the attack surface by up to 67%.

Lastly, we summarize several critical future research directions. In terms of achieving fast, deliberative planning and decision making, it is expected that both statistical and formal approaches are utilized, whereby the statistical inference and formal reasoning systems live side-by-side and interact, prompt, inform and correct each other. Integration of statistical inference and formal reasoning should be done in such a way to allow each of the methods to independently but synergistically cooperate and potentially enrich each other's knowledge base.

As future works, we not that generating test cases to cover all corner cases for formal verification is a challenging problem in general. To deal with this problem, we note that some approaches, such as fuzzing techniques [91], can be useful. In terms of improving the statistical methods, we could increase the training data size and use related machine learning optimization like cross-validation to split small data set [88] for further performance improvements. Fundamentally, deep learning approach cannot guarantee zero false positives. In this case, we can provide feedback to the training phase of deep learning module as soon as we observe false positives. This will be helpful to improve the accuracy of our function mapping module. Moreover, more complex deep learning algorithms can be further tested, such as bi-directional RNN and long-short-term memory (LSTM), which have been proven a better performance for modeling longer sequential information.

Bibliography

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

[2] SQLite. https://www.sqlite.org.

[3] IDA Pro disassembler. https://www.hex-rays.com/products/ida/, 2016.

[4] ACME Lab. Thttpd. http://www.acme.com/software/thttpd/.

[5] S. Afshan, P. McMinn, and M. Stevenson. 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*, pages 352–361. IEEE, 2013.

[6] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 263–277. IEEE, 2008.

[7] D. Alan. Pscan: A limited problem scanner for C source files. http://deployingradius.com/pscan/, 2009.

[8] M. Alhazmi, P. Dehghanian, S. Wang, and B. Shinde. Power grid optimal topology control considering correlations of system uncertainties. In *2019 IEEE/IAS 55th Industrial and Commercial Power Systems Technical Conference (I&CPS)*, pages 1–7. IEEE, 2019.

[9] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 38–49. ACM, 2015.

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

[11] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*, pages 86–95. IEEE, 1995.

[12] B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM Journal on Computing*, 26(5):1343–1362, 1997.

[13] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley. Byteweight: Learning to recognize functions in binary code. USENIX, 2014.

[14] 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)*, volume 13, page 14, 2010.

[15] 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*, pages 625–634. IEEE, 2004.

[16] 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*, pages 368–377. IEEE, 1998.

[17] C. M. Bishop. Machine learning and pattern recognition. *Information Science and Statistics. Springer, Heidelberg*, 2006.

[18] R. Bodík, R. Gupta, and V. Sarkar. ABCD: eliminating array bounds checks on demand. In *ACM SIGPLAN Notices*, volume 35, pages 321–333. ACM, 2000.

[19] R. Bodík, R. Gupta, and V. Sarkar. ABCD: eliminating array bounds checks on demand. In *ACM SIGPLAN Notices*, volume 35, pages 321–333. ACM, 2000.

[20] C. Cadar, D. Dunbar, D. R. Engler, et al. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[21] J. Chen, R. C. Chiang, H. H. Huang, and G. Venkataramani. Energy-aware writes to non-volatile main memory. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*, page 6. ACM, 2011.

[22] J. Chen and G. Venkataramani. Cc-hunter: Uncovering covert timing channels on shared processor hardware. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 216–228. IEEE, 2014.

[23] J. Chen, G. Venkataramani, and H. H. Huang. Repram: Re-cycling pram faulty blocks for extended lifetime. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12. IEEE, 2012.

[24] J. Chen, G. Venkataramani, and H. H. Huang. Exploring dynamic redundancy to resuscitate faulty pcm blocks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 10(4):31, 2014.

[25] J. Chen, Z. Winter, G. Venkataramani, and H. H. Huang. rpram: Exploring redundancy techniques to improve lifetime of pcm-based main memory. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 201–202. IEEE, 2011.

[26] J. Chen, F. Yao, and G. Venkataramani. Watts-inside: A hardware-software cooperative approach for multicore power debugging. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 335–342. IEEE, 2013.

[27] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 378–387. IEEE, 2005.

[28] 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*, pages 23–29. ACM, 2017.

[29] Y. Chen, S. Sun, T. Lan, and G. Venkataramani. Toss: Tailoring online server systems through binary feature customization. In *FEAST workshop*, 2018.

[30] G. Cokins. *Performance management: finding the missing pieces (to close the intelligence gap)*, volume 2. John Wiley & Sons, 2004.

[31] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In *European Symposium on Programming*, pages 520–535. Springer, 2007.

[32] 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*, pages 253–262. ACM, 2004.

[33] A. C. de Melo. The new linux 'perf' tools. In *Slides from Linux Kongress*, volume 18, 2010.

[34] P. Dehghanian, S. Aslan, and P. Dehghanian. Maintaining electric system safety through an enhanced network resilience. *IEEE Transactions on Industry Applications*, 54(5):4927–4937, 2018.

[35] N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *ACM SIGPLAN Notices*, volume 38, pages 155–167. ACM, 2003.

[36] 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*, pages 109–118. IEEE, 1999.

[37] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. volume 19, pages 42–51. IEEE, 2002.

[38] H. Fang, S. S. Dayapule, F. Yao, M. Doroslovački, and G. Venkataramani. A noise-resilient detection method against advanced cache timing channel attack.

In *2018 52nd Asilomar Conference on Signals, Systems, and Computers*, pages 237–241. IEEE, 2018.

[39] H. Fang, S. S. Dayapule, F. Yao, M. Doroslovački, and G. Venkataramani. Prefetch-guard: Leveraging hardware prefetches to defend against cache timing channels. In *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 187–190. IEEE, 2018.

[40] H. Fang, S. S. Dayapule, F. Yao, M. Doroslovački, and G. Venkataramani. Prodact: Prefetch-obfuscator to defend against cache timing channels. *International Journal of Parallel Programming*, 47(4):571–594, 2019.

[41] H. Fang, M. Doroslovački, and G. Venkataramani. Eraseme: A defense mechanism against information leakage exploiting gpu memory. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, pages 319–322. ACM, 2019.

[42] C. Franks, Z. Tu, P. Devanbu, and V. Hellendoorn. Cacheca: A cache language model based code suggestion tool. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 705–708. IEEE Press, 2015.

[43] 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*, pages 147–156. ACM, 2010.

[44] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer overrun detection using linear programming and static analysis. In *Proceedings of the 10th ACM conference on Computer and Communications Security*, pages 345–354. ACM, 2003.

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

[46] A. Gionis, P. Indyk, R. Motwani, et al. Similarity search in high dimensions via hashing. In *Vldb*, volume 99, pages 518–529, 1999.

[47] C. Goller and A. Kuchler. Learning task-dependent distributed representations by backpropagation through structure. In *Neural Networks, 1996., IEEE International Conference on*, volume 1, pages 347–352. IEEE, 1996.

[48] L. C. Harris and B. P. Miller. Practical analysis of stripped binary code. *ACM SIGARCH Computer Architecture News*, 2005.

[49] V. J. Hellendoorn and P. Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 763–773. ACM, 2017.

[50] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 837–847. IEEE, 2012.

[51] Y. Hu, H. Liu, and H. H. Huang. Tricore: Parallel triangle counting on gpus. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 171–182. IEEE, 2018.

[52] X. Huang, F. Alleva, H.-W. Hon, M.-Y. Hwang, K.-F. Lee, and R. Rosenfeld. The SPHINX-2 speech recognition system: an overview. volume 7, pages 137–148. Elsevier, 1993.

[53] Y. Ji, B. Bowman, and H. H. Huang. Securing malware cognitive systems against adversarial attacks. In *2019 IEEE International Conference on Cognitive Computing (ICCC)*, pages 1–9. IEEE, 2019.

[54] Y. Ji, H. Liu, and H. H. Huang. ispan: Parallel identification of strongly connected components with spanning trees. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 731–742. IEEE, 2018.

[55] 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*, pages 96–105. IEEE Computer Society, 2007.

[56] Y. Jiang, D. Wu, and P. Liu. Jred: Program customization and bloatware mitigation based on static analysis. In *Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*, volume 1, pages 12–21. IEEE, 2016.

[57] D. Jurafsky and J. H. Martin. Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition, 2009.

[58] 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*, 28(7):654–670, 2002.

[59] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 187–196. ACM, 2005.

[60] Y. Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[61] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *International Static Analysis Symposium*, pages 40–56. Springer, 2001.

[62] K. A. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1-2):77–108, 1996.

[63] T. Kroes, A. Altinay, J. Nash, Y. Na, S. Volckaert, H. Bos, M. Franz, and C. Giuffrida. Binrec: Attack surface reduction through dynamic binary recov-

ery. In *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation*, 2018.

[64] P. Kumar and H. H. Huang. Graphone: A data store for real-time analytics on evolving graphs. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pages 249–263, 2019.

[65] J. Lai, X. Lu, F. Wang, P. Dehghanian, and R. Tang. Broadcast gossip algorithms for distributed peer-to-peer control in ac microgrids. *IEEE Transactions on Industry Applications*, 55(3):2241–2251, 2019.

[66] L. Li, M. Doroslovački, and M. H. Loew. Discriminant analysis deep neural networks. In *53rd Annual Conference on Information Sciences and Systems*, 2019.

[67] L. Li, M. Doroslovački, and M. H. Loew. Loss functions forcing cluster separations for multi-class classification using deep neural networks. In *2019 Conference Record of the Fifty Third Asilomar Conference on Signals, Systems and Computers (ASILOMAR)*, 2019.

[68] P. Li, Y. Liu, and M. Sun. Recursive autoencoders for itg-based translation. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 567–577, 2013.

[69] Y. Li and T. Lan. Multichoice games for optimizing task assignment in edge computing. In *2018 IEEE Global Communications Conference (GLOBECOM)*, pages 1–7. IEEE, 2018.

[70] 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*, 11(12):2748–2762, 2016.

[71] 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*, 32(3):176–192, 2006.

[72] H. Liu and H. H. Huang. Enterprise: breadth-first graph traversal on gpus. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2015.

[73] H. Liu and H. H. Huang. Graphene: Fine-grained {IO} management for graph computing. In *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, pages 285–300, 2017.

[74] H. Liu and H. H. Huang. Simd-x: Programming and processing of graph algorithms on gpus. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 411–428, 2019.

[75] 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*, volume 5, 2005.

[76] R. McNally, K. Yiu, D. Grove, and D. Gerhardy. Fuzzing: the state of the art. Technical report, DTIC Document, 2012.

[77] T. Mikolov, S. Kombrink, A. Deoras, L. Burget, and J. Cernocky. Rnnlm-recurrent neural network language modeling toolkit. In *Proc. of the 2011 ASRU Workshop*, pages 196–201, 2011.

[78] 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*, pages 86–97. ACM, 2014.

[79] J. Ming, D. Xu, Y. Jiang, and D. Wu. Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In *Proceedings of the 26th USENIX Security Symposium. USENIX Association*, pages 253–270, 2017.

[80] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of*

the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 245–258. ACM, 2009.

[81] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. *ACM Sigplan Notices*, 44(6):245–258, 2009.

[82] M. Nazemi, M. Moeini-Aghtaie, M. Fotuhi-Firuzabad, and P. Dehghanian. Energy storage planning for enhanced resilience of power distribution networks against earthquakes. *IEEE Transactions on Sustainable Energy*, 2019.

[83] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *ACM SIGPLAN Notices*, volume 37, pages 128–139. ACM, 2002.

[84] J. Oh, C. J. Hughes, G. Venkataramani, and M. Prvulovic. Lime: A framework for debugging load imbalance in multi-threaded execution. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 201–210. ACM, 2011.

[85] T. J. Parr and R. W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[86] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

[87] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 138–157. IEEE, 2016.

[88] G. C. Smith, S. R. Seaman, A. M. Wood, P. Royston, and I. R. White. Correcting for optimistic prediction in small data sets. *American journal of epidemiology*, 180(3):318–324, 2014.

[89] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Ng, and C. Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.

[90] O. Source. Dyninst: An application program interface (api) for runtime code generation. *Online, http://www. dyninst. org*, 2016.

[91] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, 2016.

[92] Y. Sui, D. Ye, Y. Su, and J. Xue. Eliminating Redundant Bounds Checks in Dynamic Buffer Overflow Detection Using Weakest Preconditions. volume 65, 2016.

[93] S. Thummalapenta and T. Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 283–294. IEEE Computer Society, 2009.

[94] J. Turian, L. Ratinov, and Y. Bengio. Word representations: a simple and general method for semi-supervised learning. In *Proceedings of the 48th annual meeting of the association for computational linguistics*, pages 384–394. Association for Computational Linguistics, 2010.

[95] Twibright Labs. Links. `http://links.twibright.com`.

[96] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *High Performance*

Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on, pages 173–184. IEEE, 2008.

[97] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. Memtracker: An accelerator for memory debugging and monitoring. *ACM Transactions on Architecture and Code Optimization (TACO)*, 6(2):5, 2009.

[98] G. Venkataramani, C. J. Hughes, S. Kumar, and M. Prvulovic. Deft: Design space exploration for on-the-fly detection of coherence misses. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2011.

[99] 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*, pages 128–135. IEEE, 2004.

[100] S. Wang, P. Dehghanian, and Y. Gu. A novel multi-resolution wavelet transform for online power grid waveform classification. In *2019 International Conference on Smart Grid Synchronized Measurements and Analytics (SGSMA)*, pages 1–8. IEEE, 2019.

[101] 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*, pages 87–98. ACM, 2016.

[102] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 511–522. ACM, 2013.

[103] T. Würthinger, C. Wimmer, and H. Mössenböck. Array bounds check elimination for the Java HotSpot^{TM} client compiler. In *Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 125–133. ACM, 2007.

[104] M. Xu, S. Alamro, T. Lan, and S. Subramaniam. Chronos: A unifying optimization framework for speculative execution of deadline-critical mapreduce jobs. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 718–729. IEEE, 2018.

[105] 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*, pages 363–376. ACM, 2017.

[106] 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 *Poster in ACM Conference on Computer and Communications Security*, 2018.

[107] 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*, pages 413–426. Springer, 2017.

[108] H. Xue, S. Sun, G. Venkataramani, and T. Lan. Machine learning-based analysis of program binaries: A comprehensive study. *IEEE Access*, 7:65889–65912, 2019.

[109] 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*, pages 11–19. ACM, 2018.

[110] H. Xue, G. Venkataramani, and T. Lan. Clone-slicer: Detecting domain specific binary code clones through program slicing. In *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation*, pages 27–33. ACM, 2018.

[111] F. Yamaguchi. Joern: A Robust Code Analysis Platform for C/C++. `http://www.mlsec.org/joern/`, 2016.

[112] F. Yamaguchi, M. Lottmann, and K. Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 359–368. ACM, 2012.

[113] 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*, pages 499–510. ACM, 2013.

[114] F. Yao, J. Chen, and G. Venkataramani. Jop-alarm: Detecting jump-oriented programming-based anomalies in applications. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 467–470. IEEE, 2013.

[115] F. Yao, M. Doroslovacki, and G. Venkataramani. Are coherence protocol states vulnerable to information leakage? In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 168–179. IEEE, 2018.

[116] F. Yao, H. Fang, M. Doroslovački, and G. Venkataramani. Leveraging cache management hardware for practical defense against cache timing channel attacks. *IEEE Micro*, 39(4):8–16, 2019.

[117] 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*, pages 109–120. IEEE, 2017.

[118] F. Yao, G. Venkataramani, and M. Doroslovački. Covert timing channels exploiting non-uniform memory access based architectures. In *Great Lakes Symposium on VLSI*. ACM, 2017.

[119] F. Yao, J. Wu, S. Subramaniam, and G. Venkataramani. Wasp: Workload adaptive energy-latency optimization in server farms using server low-power states. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 171–178. IEEE, 2017.

[120] F. Yao, J. Wu, G. Venkataramani, and S. Subramaniam. A comparative analysis of data center network architectures. In *2014 IEEE International Conference on Communications (ICC)*, pages 3106–3111. IEEE, 2014.

[121] F. Yao, J. Wu, G. Venkataramani, and S. Subramaniam. A dual delay timer strategy for optimizing server farm energy. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 258–265. IEEE, 2015.

[122] F. Yao, J. Wu, G. Venkataramani, and S. Subramaniam. Ts-bat: Leveraging temporal-spatial batching for data center energy optimization. In *GLOBECOM 2017-2017 IEEE Global Communications Conference*, pages 1–6. IEEE, 2017.

[123] F. Yao, J. Wu, G. Venkataramani, and S. Subramaniam. Ts-batpro: Improving energy efficiency in data centers by leveraging temporal–spatial batching. *IEEE Transactions on Green Communications and Networking*, 3(1):236–249, 2018.

[124] D. Ye, Y. Su, Y. Sui, and J. Xue. WPBound: Enforcing spatial memory safety efficiently at runtime with weakest preconditions. In *Software Reliability Engineering (ISSRE), IEEE 25th International Symposium on*, pages 88–99. IEEE, 2014.

[125] M. Zalewski. American fuzzy lop, 2007.