Clone-Hunter: Accelerated Bound Checks Elimination via Binary Code Clone Detection

Anonymous Author(s)

Abstract

Memory access violation and unsafe pointer usage are the most common types of vulnerabilities in binary executables. To protect memory safety, array bound checks are inserted to detect out-of-bound accesses. Unfortunately, array bound checks contribute to high runtime overheads. Although redundant bound checks elimination techniques have been developed, they suffer from limited scalability. This is because, the number of memory bound checks are often numerous to eliminate them one-by-one.

In this paper, we propose Clone-Hunter, a practical and scalable framework for redundant bound checks elimination in binary executables. Our approach leverages *binary code clone detection* to reduce the extensive efforts in eliminating redundant bound checks. Clone-Hunter employs a bound verification mechanism using binary symbolic execution to improve the accuracy of safe removal of bound checks. Our results show Clone-Hunter can swiftly identify redundant bound checks 90× faster than pure binary symbolic execution. We note that Clone-Hunter achieves similar removal ratio for redundant bound checks as prior approaches, in addition to achieving several orders of magnitude improvement in time-to-solution (the time spent to remove redundant bound checks).

1 Introduction

Memory related bugs and buffer overflows are oft-cited problems leading to software security issues. This concern is exacerbated in legacy applications where only binary executables are available, and have been in deployment for a number of years in production systems. Numerous instances of such legacy binary code exist in domains such as airspace, military and banking [27]. Illegal memory accesses and unsafe pointer usage in such applications can lead to compromising sensitive user data. We note that memory safety in applications continues to remain as a major concern. For instance, in August 2017, Microsoft identified a flaw in the legacy JET database program supported by Windows 7 and 10 editions. This bug was reported to have the potential to take over users private computer full system control remotely [40].

To secure and protect binary executables from memory and pointer-related problems, techniques that ensure safety through checking array bounds have been developed [10, 24, 35]. However, these techniques and tools still incur high runtime overheads when they are performed exhaustively especially when such checks turn out to be redundant for most benign pointer accesses. Note that such overheads can be

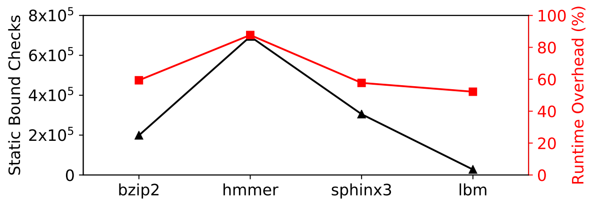


Figure 1. Total static bound checks and runtime overhead incurred by bound checks instrumented applications over uninstrumented applications as baseline in SPEC2006 Benchmarks

dramatically higher in pointer-intensive programs. Figure 1 shows the total number of static bound checks and runtime overheads introduced by an array bound checker tool, Softbound [24] for several representative SPEC2006 benchmarks. In order to improve and reduce such high performance overheads, *redundant bound check elimination* approaches have been developed [6, 41, 46]. By eliminating redundant array bound checks, the performance overheads resulting from such unnecessary checks can be avoided. However, note that such *redundant check elimination methods still need to analyze every single pointer deference to compute the constraints involving pointer-related variables, and verify whether bound checks are redundant and be removed effectively from that location*. In case of applications involving billions of pointer dereferences, the task of verifying the redundancy of bound check can still be prohibitively expensive or impossible in practice.

Our work is motivated by the key observation that software applications usually have an abundant number of similar code fragments, called *code clones* [19, 20]. 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 [18]). *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 to identify code clones and reduce the time-to-solution in terms of eliminating redundant bound checks in binaries.

We propose a novel approach, *Clone-Hunter*, in order to perform rapid elimination of redundant bound checks in binary applications through identifying *identical* code clones. Clone-Hunter first finds all of the identical clone pairs in binaries, and forms clusters of such clones. It then picks a random seed sample from each cluster, and with the help of a binary symbolic executor, determines 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 the 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.

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. This approach can be especially significant in binary applications where memory safety is important to system security while making sure that the performance of such systems is not adversely affected. 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[9].

In summary, the contributions of this paper are:

1. We propose *Clone-Hunter*, a framework that leverages machine learning to replicate the decision to remove array bound checks on identical code clones, thereby reducing the time-to-solution in terms of eliminating redundant bound checks in application binaries.

2. We demonstrate a novel use of joint statistical-formal learning where safe removal of redundant bound checks are identified using binary symbolic execution, and machine learning-based clone detectors are used in accelerating the elimination of redundant checks.

3. We implement a prototype of Clone-Hunter and evaluate using real-world applications from SPEC2006 benchmarks suite [1]. Our results show the time-to-solution (time spent to remove bound checks) for Clone-hunter is 90× faster compared to pure binary symbolic execution, and three out of four applications experienced time-out when pure binary symbolic executors are used.

The rest of this paper is organized as follows: Section 2 gives the overview of our approach along with some technical details. In Section 3, we illustrate how we design and implement our system, respectively. We evaluate our system for redundant bound checks elimination and state the results in Section 4. Section 5 and 6 give some related works and conclusion of this paper.

2 Approach Overview

In this section, we give an overview how Clone-Hunter accelerates the removal of redundant bound checks in binary executables. The main components of Clone-Hunter is shown in Figure 2.

For given binaries that are instrumented with bound checks, Clone-Hunter first employs a Binary Code Clone Detector to identify identical code clone pairs. We disassemble binary executables and work with the resulting assembly code. In order to detect code clones, the assembly code is transformed into normalized instruction sequences with intermediate representations in order to remove instruction-specific details, such as register names and memory addresses. This step improves the performance of machine learning algorithms and enables Clone-Hunter to find clones that are syntactically identical. Note that our clones will very likely be semantically equivalent as well since two identical instruction patterns will very likely perform the same *functionally logical operation*. This is further verified through binary symbolic execution later. Then, we generate feature vectors for each normalized instruction sequence, embed them into vector space and use clustering algorithms to find code clones (more details in Section 3.1.1). Note that the detected code clones need to be consolidated, because there could be overlapping and duplicated clones due to sliding window algorithm, and because we only need to consider pointer-related code (since bound checks relevant to only pointers). Thus, the code clones that are duplicated or not pointer-related would be removed from further consideration, during code clone consolidation (Section 3.1.3).

The next step is to use binary symbolic execution to verify whether redundant bound checks can be safely removed. This process is performed as follows: We sample each cluster of code clones and apply binary symbolic execution to determine whether array bound check is possible on the selected samples. Array bound checks will be redundant if the pointer dereference is guaranteed to be safe and never out of array bounds. Since all code clones in the same cluster are semantically equivalent, we replicate the decision of array bound checks removal on all code clones in the cluster. Note that code clone detection through clustering algorithms are not guaranteed to be precise, and hence we need to further verify the validity of clone detection, by selecting a random subset of samples within the cluster and performing binary symbolic execution on all of them.

Finally, we perform redundant bound checks elimination using binary rewriting to remove the corresponding bound check instructions. Section 3.3 describes our implementation of this module in more detail.

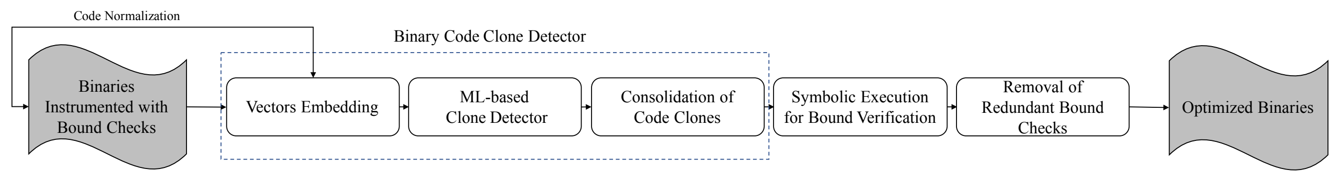
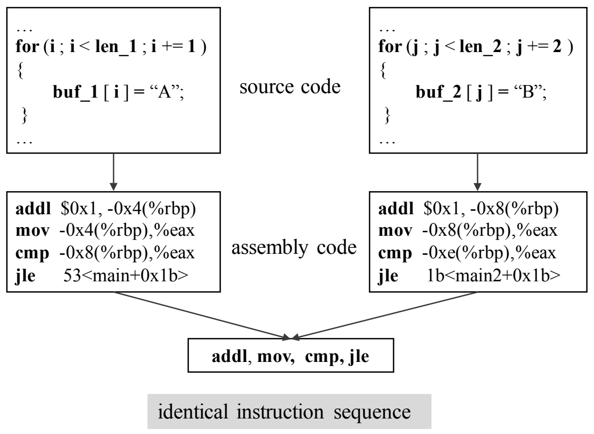

Code Normalization
Binary Code Clone Detector
Binaries Instrumented with Bound Checks
Vectors Embedding
ML-based Clone Detector
Consolidation of Code Clones
Symbolic Execution for Bound Verification
Removal of Redundant Bound Checks
Optimized Binaries

Figure 2. Approach Overview



```
...
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. Motivation example for code normalization

```
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 Assembly Code   Normalized Assembly Code

Figure 4. An example illustrating normalization of given code region.

3 System Design and Implementation

In this section, we present details of our Clone-Hunter and show how our system is implemented.

3.1 Binary Code Clone Detection

Clone-Hunter accelerates redundant bound checks removal by identifying binary code clones and replicating the same removal condition on these code clones.

3.1.1 Vector Embedding

We first disassemble the target binaries, and detect code clones in the assembly code, which are syntactically or semantically identical. 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 identical code clones if they are exactly the same except for some certain constant values, offsets in memory locations, or particular addresses used as branch targets. For example, Figure 3 shows two identical source code snippets and their corresponding assembly code. As we can see, their assembly codes share the same instruction sequence but with different operands. To address these issues, 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 code clone detection using clustering algorithms.

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 under 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 and functionality, but 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 4 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 feature vector, denoted as $C_i = (C_{i_1}, C_{i_2}, ..., C_{i_n})$, where C_{i_k} (for $k = 1, \dots, n$) measures the occurrence of normalized instruction k in code region i. This process is illustrated in Figure 5 for the code region example shown in Figure 4.

In Clone-Hunter, we employ IDA Pro [3] binary disassembler 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).

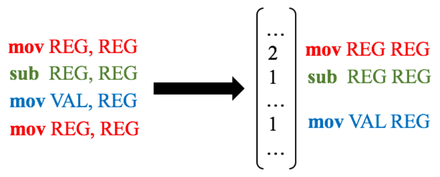


Figure 5. Normalized assembly code embedded into vector space

3.1.2 Machine Learning based Clone Detector

After embedding 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 being provided in advance. 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, via their Negative Squared Euclidean distance, i.e.,

$$S(i, j) = -||C_i - C_j||_2^2. \tag{1}$$

where $|| \cdot ||_2$ denotes the L-2 vector norm. On the other hand, the diagonal values $S(k, k)$ are algorithm 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 to C_k.

In the AP algorithm, $R(i, k)$ is sent from data points to candidate cluster centers, and the *Availability matrix* $A(i, k)$ is sent from candidate cluster centers to data points. 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 an zero Availability matrix $A(i, k)$, and in each iteration, updates both $R(i, k)$ and $A(i, k)$ in a coupled fashion, according to

$$R(i, k) = S(i, k) - \max_{k':k' \neq k} \{A(i, k') + S(i, k')\} \tag{2}$$

$$A(i, k) = \min\{0,\ R(k, k) + \sum_{i' \notin \{i,k\}} \max\{0,\ R(i', k)\}\} \tag{3}$$

Note that the $A(i, k)$ is non-positive due to Equation (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 responsibilities point 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 also stop changing. More details about AP clustering can be found in [12].

We implement our clustering-based code clone detector in Python using a machine learning tool Scikit-learn [25]. We instrument its AP clustering API - *sklearn.cluster* for our clustering module.

3.1.3 Consolidation of Code Clones

Code Clone Consolidation removes duplicated and pointer-irrelevant code clones from further consideration.

We first filter out the pointer-irrelevant code clones by checking if they contain bound check-related instructions. For example, Softbound-instrumented bound checks instructions will contain "*softbound_spatial_checks*" symbol in binary executables. This enables filtering out these instructions using such symbols.

When using sliding window algorithm to generate code regions, it can create overlapping windows and thus result 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 line numbers 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-size, continuous code clone, $(s, e) \cup (s', e')$. This consolidation procedure is performed until all consolidated code clones are non-overlapping.

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

3.2 Symbolic Execution for Bound Verification

Clone-Hunter utilizes clustering algorithms in Machine Learning to identify binary code clone to assist redundant bound checks removal. *Based on our observations from large scale of code samples, It is highly likely that the redundant bound checks in two code samples can be both removed if they are identical code clones.* To formally verify if the code clones detected by Clone-Hunter guarantee simultaneous bound checks removal, we utilize binary symbolic execution.

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

1. **Redundant bound checks identification:** First, we pick a random code clone sample as *seed clone sample* in each cluster to verify if the bound checks are redundant. This can be done by determining if the pointer dereference is safe, and that no memory violation can exist. We deploy binary symbolic execution to execute such set of *seed clone sample* and determine whether the bound checks are redundant based on the outputs

from symbolic execution. Since the seed clone sample is just a code snippet from the original program, we perform partial symbolic execution starting at the beginning of the seed clone sample to the end of seed clone sample based on instruction addresses. To deal with the incomplete program state while performing partial symbolic execution, we should make the values of unknown variables in this code region as symbolic variables instead of concrete values. Then, we are able to perform arithmetic operations with symbolic variables through symbolic execution. If the pointers in *seed clone sample* turn out to be safe, then we continue to execute the next step of bound verification. If not safe, we terminate bound verification procedure and apply the decision as "Not redundant" to the other corresponding identical code clones in the cluster to keep bound checks.

2. **Verification of bound identification:** Clustering algorithm cannot offer a 100% guarantee in terms of ensuring simultaneous, safe bound check removal from all detected code clones. It is possible that two code snippets are found to be identical code clones, but have different bound safety conditions and do not allow simultaneous bound checks removal. To further improve the removal's accuracy, we select a random set of code clone samples within the same cluster and perform the same procedure using binary symbolic execution to check whether the bound checks removal conditions on these code clones are indeed identical.
3. **Applying Removal Decision:** If the random code clones samples turn out to be safe as the *seed clone sample* does, then we apply the final decision as "Redundant" to the all corresponding identical code clones in the cluster to remove bound checks. Nevertheless, if the random code clones samples turn out to be not safe, which is against with the decision we made for *seed clone sample* from the previous step, then we apply the final decision as "Not redundant" to all clone samples in the cluster instead.

We instrument a binary analysis framework angr [32] 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 the total number of instructions in the code region.

3.3 Removal of Redundant Bound Checks

To delete instructions in binary executables, we deploy a Static Binary Rewriting tool Dyninst [34]. As we mentioned in previous section, we store additional information for each code region including their start and end addresses. We use such address information to rewrite control transfers. We implement our Bound Checks Remover in C++ with Dyninst. Given a code clone as input, we scan each instruction and remove bound checks. We obtain Optimized Binaries as output.

4 Evaluation

4.1 Experiment Setup

We selected 4 different real-world applications: bzip2, hmmer, lbm and sphinx3 from SPEC2006 benchmark suite [1] and use the largest *reference* inputs provided with SPEC benchmark. We manually injected buffer overflow bugs to verify whether there are any false positives. Some techniques such as fault injection [15, 39] can be applied in larger scale programs for future work. 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 checks tool: Softbound [24] to illustrate our mechanism and create bound checks in binary executables.

Bench.	#Total Static Instructions	#Clusters	#Cloned Instructions	%Cloned Instructions
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 1. Binary Code Clone Detection Statistics

4.2 Effectiveness of Binary Code Clone Detection

We evaluated our binary code clone detector with different window size and stride. The number of cloned instructions shows the code clone coverage over the whole program. We noticed that there are more code clones detected with smaller window size in general, but not always the case. If the window size is small, there will be more noise, such as a clone with only one *nop* instruction. Here, we observed that we detect the most code clones with maximum window size equals to 100 (minimum window size = 2) and stride equals to 4. For simplicity, we represent the statistics of code clones with fixed window size range and stride as from 2 to 100 and 4 respectively. Table 1 shows for each benchmark, how many static instructions, clone clusters, cloned instructions, and percentage of cloned instructions over total static instructions generated by our machine learning based clone detector.

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 machine learning algorithms.

4.3 Overhead of Binary Symbolic Execution

We evaluated the overhead of binary symbolic executors for bound verification, and compared with Pure Symbolic Execution over entire binary programs. We instrumented a binary analysis framework angr [32] as our baseline.

Bench.	**Application Type**	**Program Size** (Byte)	**Pure Symbolic Execution time Whole Program** (sec)	**Pure Symbolic Execution time Function-Level** (sec)	**Clone-Hunter assisted Symbolic Execution time** (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 2. Comparison of symbolic execution time spent in Clone-Hunter and Pure Symbolic Execution

Table 2 presents the runtime overhead from pure symbolic execution and Clone-Hunter. We evaluate pure symbolic execution overhead on whole program and conduct partial symbolic execution on each function as function-level overhead. We set up 43200 seconds (12 hours) as TIME OUT in this experiment. In this experiment, we set up a threshold for bound verification. Since the smallest cluster contains only 2 code clone samples, we chose a lower bound as 2 code clone samples for bound verification. For larger clusters, we pick 30% sampling rate as upper bound to random select code clone samples (noting that this sampling rate is tunable depending on the users). For example, we select 6 different code clone samples if one cluster contains 20 total code clones. The time spent in Clone-Hunter assisted Symbolic Execution is calculated as the summation of symbolic execution time in the random seed clones in each cluster. We observe that Clone-Hunter always spends less time than angr in the terms of Symbolic Execution overhead. Notably, for bzip2, sphinx3 and hmmer, angr fails to finish the execution within TIME OUT. The time-to-solution (the time spent to remove bound checks) for Clone-hunter is 90× faster compared to pure Binary Symbolic Execution in lbm. On the other hand, it is easy to see that why sphinx3 takes more time in Clone-Hunter. Our code clone detector detected the number of cluster as 2,771 in sphinx3, 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 entire program cost much more time in angr, that is because pure exhaustive approach is going to Symbolic Execute the entire program, which can lead to problems like path explosion. Some functions in bzip2 contain more loop operations and function calls, it leads to a longer symbolic execution time in entire program analysis, sphinx3 and hmmer have a similar program behavior during entire program symbolic execution.

4.4 Redundant Bound Checks Elimination

We evaluated our approach in identical binary code clones for redundant bound checks elimination. Figure 6 plots the comparison of Softbound runtime execution overhead before and after using Clone-Hunter to eliminate redundant bound checks at runtime. We further evaluate the percentage false positives of removing redundant bound checks (where a false positive occurs if a bound check identified as redundant is indeed necessary and cannot be safely removed). Clone-Hunter achieves an average reduction of 34.24% compared to Softbound runtime overheads.

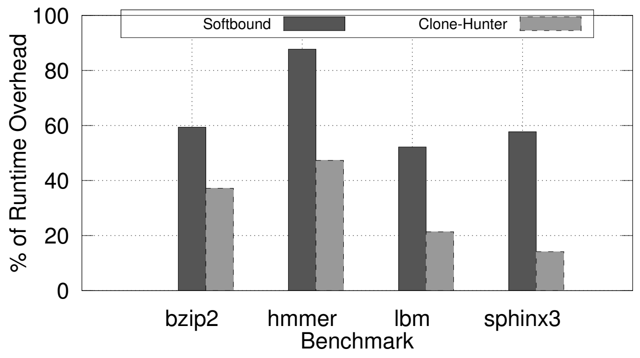


Figure 6. Runtime overhead of softbound-instrumented applications and Clone-Hunter optimized bound checks.

Bench.	bzip2	hmmer	lbm	sphinx3
%Dynamic Checks Removed	26.72%	42.31%	30.90%	45.54%
%False Positive	0.00%	0.00%	0.00%	0.00%

Table 3. Percentage of Softbound Dynamic Bound Checks removed by our approach

Table 3 shows the percentage of dynamic bound checks eliminated in all 4 benchmarks, along with rate of false positive reported under Clone-Hunter. Clone-Hunter shows an average 36.37% redundant bound checks elimination ratio, with the highest 45.54% at sphinx3, which is similar to other existing source code based redundant bound checks approaches, such as SIMBER [41]. However, current approaches still need the access to source code. *To the best of our knowledge, Clone-Hunter is the first framework for binary bound checks removal.* Also, the result represents that Clone-Hunter obtains zero false positive in all benchmarks used in this evaluation.

Note that the percentage of dynamic checks removed by our approach is not linearly related to the number of Softbound runtime overhead. We further analyzed the breakdown of Softbound execution time. We note that Softbound load instruction dereference check has 4× runtime slowdown compared to the corresponding store instruction dereference 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 deference checks. As we can see, sphinx3 achieves the highest overhead reduction performance than others. We further analyzed sphinx3 and we found Clone-Hunter removes 62.33% load instruction related checks of the total Softbound dereference bound checks. We also notice that our approach outperforms the performance of source code-based redundant bound removal methods [41]. 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 to expand such macros and unroll the loops within them.

5 Related Work

Prior solutions for eliminating redundant bound checks are usually based on static source code analysis. WPbound [46] and ABCD [6] both reduce redundant bound checks in Softbound [24] by solving a system of linear inequalities obtained through static code analysis. In contrast, SIMBER [41] proposes a learning approach based on runtime statistics to refine the bound elimination conditions. These methods are often limited in their scalability due to the need to derive bound elimination conditions and to analyze every single bound check locations. In addition, these techniques are only applicable to software systems whose source code is available.

To protect memory safety in software systems, various static code analysis [14, 23, 42, 48], have been developed to analyze program behaviors and to identify bug/vulnerabilities, such as SECRET [47], StatSym [45], HOTracer [17] and Sarre [21]. These techniques also suffer from growing program size, since the amount of analysis required is directly proportional to the sizes of software systems. Another line of works propose hardware based array bound checks for memory protection [37]. For instance, MemTracker [36] provides hardware support to accelerate array bound checks. Shen et al. [31] present a hardware based framework for flexible and fine-grain heap memory protection. Such techniques can efficiently support proper array bound checks and violations using hardware support, but they may involve hardware modifications and hardware costs.

To address the scalability issue, a number of tools have been developed for source-code clone detection [5, 18, 19, 30, 38]. In particular, Chunky [43] uses context-based Natural Language Processing for static code analysis. These techniques, often relying on source code or intermediate representation, are intended to identify general code clones. Another line of work is detecting code clones in binaries. Pewny et al. [26] has developed a prototype for binary code clone detection through translating the binary code to an intermediate representation. A binary code clone framework is proposed by Yikun et al. [16] with a more advanced approach using Deep Learning techniques to identify identical code clones within different complied architectures and configurations. While in this paper we harness code clone detection and formal analysis techniques in an integrated framework to enable rapid bound elimination at scale. We current only work on identical code clones. Thus, our code clone detector is efficient, such binary code clone approaches can be applied in the future to deal with problems like cross-platforms and semantically equivalent code clone detection. Instead of deploying code clone detection, some other techniques, such as using a pre-trained probabilistic model to extract code features and track similar code fragments [29].

Binary code analysis is particularly important for legacy software systems, whose source codes are often not available. Several static binary analysis tools have been developed to support the safety of lower-level binaries, such as rev.ng [11], vfGuard [28], ByteWeight [4] and BitBlaze [33]. Over the past decade, lots of efforts have been made in binary reverse engineering. Caballero et al.[7] summarized various binary code type inference and binary analysis methods for improving program security, such as binary differing for vulnerability detection, binary program customization through binary rewriting and program patching against software obfuscation using tools like BinSim [22], DamGate [8] and BinHunt [13]. In another line of work, Jop-alarm [44] uses binary analysis to track indirect jumps for detecting jump oriented program attack.

6 Conclusion and Future Work

In this paper, we present a novel framework, Clone-Hunter, integrating Machine Learning based binary code clone detection with redundant bound checks elimination in binary executables. We evaluated our approach in 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. Currently, Clone-Hunter only works for identical code clones to remove bound checks. As future work, we will analyze Clone-Hunter on code patterns that are not exact matches that still semantically equivalent.

References

[1] 2006. SPEC CPU 2006. https://www.spec.org/cpu2006/. (2006).
[2] 2010. SQLite. https://www.sqlite.org. (2010).

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

[4] Tiffany Bao, Johnathon Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. Byteweight: Learning to recognize functions in binary code. USENIX.

[5] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*. IEEE, 368–377.

[6] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. 2000. ABCD: eliminating array bounds checks on demand. In *ACM SIGPLAN Notices*, Vol. 35. ACM, 321–333.

[7] Juan Caballero and Zhiqiang Lin. 2016. Type inference on executables. *ACM Computing Surveys (CSUR)* 48, 4 (2016), 65.

[8] Yurong Chen, Tian Lan, and Guru Venkataramani. 2017. DamGate: Dynamic Adaptive Multi-feature Gating in Program Binaries. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*. ACM, 23–29.

[9] Gary Cokins. 2004. *Performance management: finding the missing pieces (to close the intelligence gap)*. Vol. 2. John Wiley & Sons.

[10] Dinakar Dhurjati and Vikram Adve. 2006. Backwards-compatible array bounds checking for C with very low overhead. In *Proceedings of the 28th international conference on Software engineering*. ACM, 162–171.

[11] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. rev. ng: a unified binary analysis framework to recover CFGs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*. ACM, 131–141.

[12] Brendan J Frey and Delbert Dueck. 2007. Clustering by passing messages between data points. *science* 315, 5814 (2007), 972–976.

[13] Debin Gao, Michael Reiter, and Dawn Song. 2008. Binhunt: Automatically finding semantic differences in binary programs. *Information and Communications Security* (2008), 238–255.

[14] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2013. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*. ACM, 45–54.

[15] Mei-Chen Hsueh, Timothy K Tsai, and Ravishankar K Iyer. 1997. Fault injection techniques and tools. *Computer* 30, 4 (1997), 75–82.

[16] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2017. Binary code clone detection across architectures and compiling configurations. In *Proceedings of the 25th International Conference on Program Comprehension*. IEEE Press, 88–98.

[17] Xiangkun Jia, Chao Zhang, Purui Su, Yi Yang, Huafeng Huang, and Dengguo Feng. 2017. Towards Efficient Heap Overflow Discovery. (2017).

[18] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 96–105.

[19] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670.

[20] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. 2005. An empirical study of code clone genealogies. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 187–196.

[21] Yongbo Li, Fan Yao, Tian Lan, and Guru Venkataramani. 2016. Sarre: semantics-aware rule recommendation and enforcement for event paths on android. *IEEE Transactions on Information Forensics and Security* 11, 12 (2016), 2748–2762.

[22] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking. In *26th USENIX Security Symposium*.

[23] Lili Mou, Ge Li, Yuxuan Liu, Hao Peng, Zhi Jin, Yan Xu, and Lu Zhang. 2014. Building program vector representations for deep learning. *arXiv preprint arXiv:1409.3358* (2014).

[24] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. *ACM Sigplan Notices* 44, 6 (2009), 245–258.

[25] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12, Oct (2011), 2825–2830.

[26] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-architecture bug search in binary executables. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 709–724.

[27] David A Powner. 2016. Federal agencies need to address aging legacy systems. *Testimony before the Committee on Oversight and Government Reform, House of Representatives* (2016).

[28] Aravind Prakash, Xunchao Hu, and Heng Yin. 2015. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries.. In *NDSS*.

[29] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from big code. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 111–124.

[30] Chanchal K Roy, James R Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming* 74, 7 (2009), 470–495.

[31] Jianli Shen, Guru Venkataramani, and Milos Prvulovic. 2006. Tradeoffs in fine-grained heap memory protection. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*. ACM, 52–57.

[32] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 138–157.

[33] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A new approach to computer security via binary analysis. *Information systems security* (2008), 1–25.

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

[35] Andrew Suffield. 2003. Bounds Checking for C and C++. *BEng dissertation, Imperial College London* (2003).

[36] Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. 2009. MemTracker: An accelerator for memory debugging and monitoring. *ACM Transactions on Architecture and Code Optimization (TACO)* 6, 2 (2009), 5.

[37] Guru Prasadh V Venkataramani. 2009. *Low-cost and efficient architectural support for correctness and performance debugging*. Georgia Institute of Technology.

[38] Vera Wahler, Dietmar Seipel, J Wolff, and Gregor Fischer. 2004. Clone detection in source code by frequent itemset techniques. In *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*. IEEE, 128–135.

[39] Jiesheng Wei, Anna Thomas, Guanpeng Li, and Karthik Pattabiraman. 2014. Quantifying the accuracy of high-level fault injection techniques for hardware faults. In *Dependable Systems and Networks, 44th Annual IEEE/IFIP International Conference on*. IEEE, 375–382.

[40] Zack Whittaker. 2017. Microsoft fixes 'critical' security bugs affecting all versions of Windows. http://www.zdnet.com/article/critical-security-bugs-affect-all-windows-versions. (2017).

[41] Hongfa Xue, Yurong Chen, Fan Yao, Yongbo Li, Tian Lan, and Guru Venkataramani. 2017. SIMBER: Eliminating Redundant Memory

Bound Checks via Statistical Inference. In *Proceedings of the IFIP International Conference on Computer Security*. Springer.

[42] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. 2012. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 359–368.

[43] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. 2013. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 499–510.

[44] Fan Yao, Jie Chen, and Guru Venkataramani. 2013. Jop-alarm: Detecting jump-oriented programming-based anomalies in applications. In *Computer Design, IEEE 31st International Conference on*. IEEE, 467–470.

[45] Fan Yao, Yongbo Li, Yurong Chen, Hongfa Xue, Venkataramani Guru, and Tian Lan. 2017. StatSym: Vulnerable Path Discovery through Statistics-guided Symbolic Execution. In *Proceedings of 47th IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE.

[46] Ding Ye, Yu Su, Yulei Sui, and Jingling Xue. 2014. WPBound: Enforcing spatial memory safety efficiently at runtime with weakest preconditions. In *Software Reliability Engineering (ISSRE), IEEE 25th International Symposium on*. IEEE, 88–99.

[47] Mingwei Zhang, Michalis Polychronakis, and R Sekar. 2017. Protecting COTS Binaries from Disclosure-guided Code Reuse Attacks. (2017).

[48] Alice X Zheng, Michael I Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. 2006. Statistical debugging: simultaneous identification of multiple bugs. In *Proceedings of the 23rd international conference on Machine learning*. ACM, 1105–1112.