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

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

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

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

Abstract

Unsafe pointer usage and illegitimate memory accesses are prevalent bugs in software. To ensure memory safety, conditions for array bound checks are inserted into the code to detect out-of-bound memory accesses. Unfortunately, these bound checks contribute to high runtime overheads, and therefore, redundant array bound checks should be removed to improve application performance.

In this paper, we propose Clone-Hunter, a practical and scalable framework for redundant bound check elimination in binary executables. Clone-Hunter first uses *binary code clone detection*, and then employs bound safety verification mechanism (using binary symbolic execution) to ensure sound removal of redundant bound checks. Our results show the Clone-Hunter can swiftly identify redundant bound checks about 90× faster than pure binary symbolic execution, while ensuring zero false positives.

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

Keywords Memory safety, Array bound checks, Machine learning, Binary analysis

ACM Reference Format:
Hongfa Xue, Guru Venkataramani, and Tian Lan. 2018. Clone-Hunter: Accelerated Bound Checks Elimination via Binary Code Clone Detection. In *Proceedings of 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3211346.3211347

MAPL, 2018, Philadelphia, PA

ACM ISBN 978-1-4503-5834-7/18/06…$15.00
https://doi.org/10.1145/3211346.3211347

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 [25]. 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 [36].

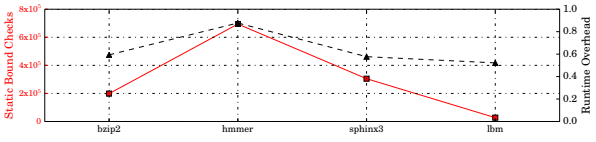

Figure 1. Number of array bound checks and the corresponding runtime overhead incurred for instrumented applications from SPEC 2006 benchmark suite [1]. Non-instrumented applications are used as baseline.

To secure and protect binary executables from memory and pointer-related problems, techniques that ensure safety through checking array bounds have been developed [10, 22, 32]. However, these techniques and tools still incur high runtime overheads when they are performed exhaustively and such checks turn out to be redundant for most benign pointer accesses. Note that such overheads can be 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 [22] for several representative SPEC2006 benchmarks. In order to reduce such high performance overheads, *redundant bound check elimination* approaches have been developed [6, 37, 42]. 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 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 checks 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* [17, 18]. 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 [16]). *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.

We propose a novel approach, *Clone-Hunter*, in order to perform 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.

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[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. In Section 3, we illustrate how we design and implement our system, respectively. We evaluate Clone-Hunter and show our experimental results in Section 4. Section 5 and 6 discusses some related works and our conclusions.

2 Approach Overview

In this section, we give an overview on 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 a given application binary that is instrumented with array bound checks, Clone-Hunter first employs a Binary Code Clone Detector to identify 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 at modest runtime overheads. Note that our clones are *more likely than not* to be functionally equivalent as well since two identical instruction patterns will very likely perform the same *functionally logical operation.* This equivalence is further verified through binary symbolic execution later.

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

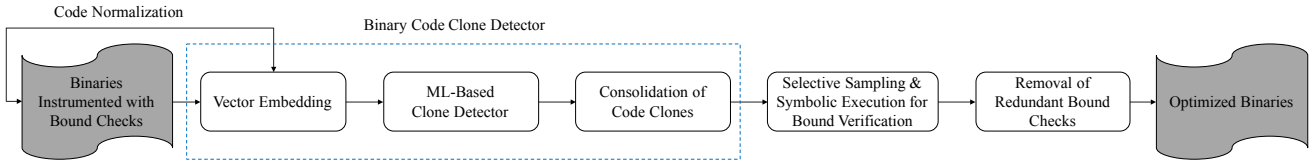

Figure 2. Overview of Clone-Hunter

need to be consolidated, because there could be overlapping and duplicated clones as we adopt a sliding window based analysis approach. Also, we only need to consider pointer-related code (since bound checks are only relevant to 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 checks are absolutely needed 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 functionally equivalent, we replicate the decision of array bound check 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 *further verify the validity of clone detection by selecting a random subset of samples* within the cluster and through performing *binary symbolic execution on all of them.*

Finally, we perform redundant bound check elimination using binary rewriting to remove the corresponding bound check instructions inserted by an array bound checker tool such as Softbound. Section 3.3 describes our implementation of this module in more detail.

3 System Design and Implementation

In this section, we present the design details of our Clone-Hunter framework, 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 replicates the decision to perform removal of bound checks on the corresponding code clones.

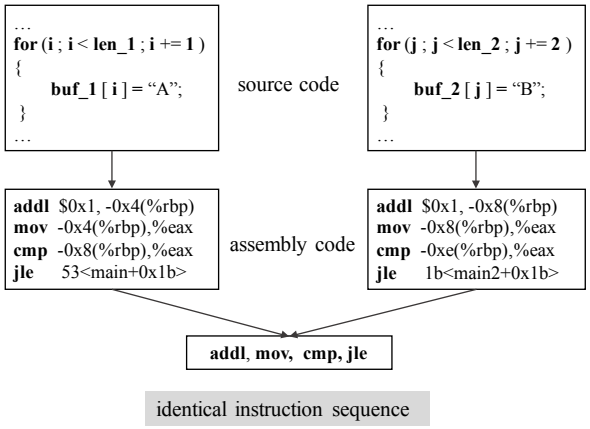

Figure 3. Motivation example for code normalization

3.1.1 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 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.

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,

```
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 4. An example illustrating normalization for given a given binary code.

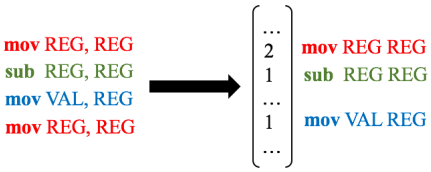

Figure 5. Normalized assembly code embedded into vector space

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 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 $k = 1, \ldots, n$) 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 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).

3.1.2 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.

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

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

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

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

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 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 [23]. We instrument its AP clustering API - *sklearn.cluster* for our clustering module.

3.1.3 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-instrumented 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 3.1.1, 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 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 [29] 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.

3.3 Removal of Redundant Bound Checks

To delete instructions in binary executables, we deploy a Static Binary Rewriting tool Dyninst [31]. 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.

4 Evaluation

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

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* 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 [22] that inserts array bound checks into application's binary executable files.

4.2 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 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.

4.3 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 [29].

Table 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. 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× 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.

4.4 Redundant Bound Checks Elimination

Figure 6 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 1. Binary Code Clone Statistics

Benchmark	#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 2. Comparison of time spent in Clone-Hunter and Pure Symbolic Execution

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

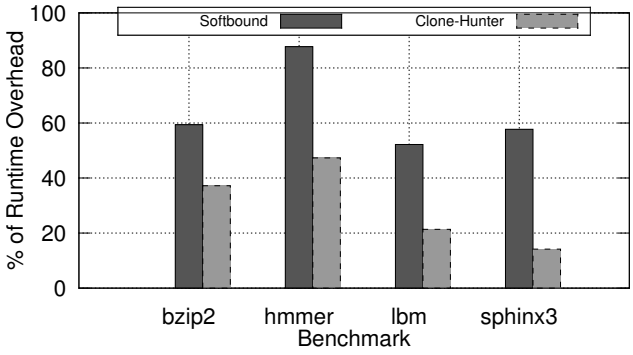

Figure 6. Runtime overhead of softbound-instrumented applications and Clone-Hunter. The baseline is non-instrumented applications.

Table 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 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 [37], 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× 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 [37], to expand such macros and unroll the loops within them.

Table 3. Percentage of Softbound's dynamic array bound checks removed by Clone-Hunter

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%

5 Related Work

Prior solutions for eliminating redundant bound checks are usually based on static source code analysis. WP-bound [42] and ABCD [6] both reduce redundant bound checks in Softbound [22] by solving a system of linear inequalities obtained through static code analysis. In contrast, SIMBER [37] 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 location. 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 methods [13, 21, 38, 44], have been developed to analyze program behavior, prior work have also studied bug/vulnerabilities, such as SECRET [43], StatSym [41], HOTracer [15] and Sarre [19]. These techniques suffer scalability issues resulting from growing program size, since the amount of analysis required is directly proportional to the sizes of software systems. Another line of work have proposed hardware-based array bound checks for memory protection. For instance, MemTracker [33, 34] provides hardware support to accelerate array bound checks. Shen et al. [28] 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 also involve hardware modifications and the associated costs.

To address the scalability issue, a number of tools have been developed for source-code clone detection [5, 16, 17, 27, 35]. In particular, Chucky [39] uses context-based Natural Language Processing for static code analysis. These techniques, often rely on source code or intermediate representation, and are intended to identify general code clones. Pewny et al. [24] has developed a prototype for binary code clone detection through translating the binary code to an intermediate representation. Yikun et al. [14] use advanced Deep Learning techniques to identify identical code clones within different complied architectures and configurations. In this paper, we harness the power of code clone detection and formal analysis techniques in an integrated framework to enable rapid bound elimination on binaries at scale. Our code clone detector is efficient, and can be applied in the future to deal with problems like cross-platforms and semantically equivalent code clone detection.

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 [26], ByteWeight [4] and BitBlaze [30]. Over the past decade, binary reverse engineering has been widely studied. Caballero et al.[7] summarized various binary code type inference and analysis methods for improving program security.

Program customization has been studied through rewriting and program patching against software obfuscation using tools like BinSim [20], DamGate [8] and BinHunt [12]. Jop-alarm [40] performs runtime analysis on binaries to track indirect jumps and detect jump oriented program-based attacks.

6 Conclusions and Future Work

In this paper, 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× 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.

Acknowledgments

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

References

[1] 2006. SPEC CPU 2006. https://www.spec.org/cpu2006/.
[2] 2010. SQLite. https://www.sqlite.org.
[3] 2016. IDA Pro disassembler. https://www.hex-rays.com/products/ida/.
[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] Debin Gao, Michael Reiter, and Dawn Song. 2008. Binhunt: Automatically finding semantic differences in binary programs. *Information and Communications Security* (2008), 238–255.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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*.

[21] 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).

[22] 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.

[23] 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.

[24] 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.

[25] 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).

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

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

[33] 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.

[34] Guru Venkataramani, Brandyn Roemer, Yan Solihin, and Milos Prvulovic. 2007. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*. IEEE, 273–284.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

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

[44] 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.