Detect Compiler Inserted Run-time Security Checks in Binary Software

Koyel Pramanick and Prasad A. Kulkarni

EECS, University of Kansas, KS, USA
{koyel_pramanick25,prasadk}@ku.edu

Abstract. Our goal in this work is to develop a mechanism to determine the presence of targeted compiler-based or automated rules-based run-time security checks in any given binary. Our generalized approach relies on several key insights. First, instructions added by automated checks likely follow just one or only a few fixed patterns or templates at every insertion point. Second, any security check will guard some *interesting* or vulnerable program structure, like return addresses, indirect jumps/calls, etc., and the placement of the security check will inform about the nature of the check. By contrast, we would not expect ordinary user code to follow any single pattern at every such interesting program location. Our technique to detect automated security checks in binary code does not rely on known code signatures that can change depending on the language, the compiler, and the security check. We implement and evaluate our technique, and present our results, observations, and challenges in this work.

Keywords: Program binary · Security check · Automated security assessment.

1 Introduction

Most software available to ordinary customers are shipped *without* any quality control indicators or metrics regarding the security and reliability properties of that software. Likewise, software are often shipped and distributed in their binary form without access to the original source code, which makes it especially challenging for customers to independently verify their security properties. End-users have limited means and few accessible tools to validate binary software security. This state of security validation for software customers persists even as the number of reported vulnerabilities have been increasing in number and severity for many years [11] and software vulnerabilities have been found to cause many disastrous real-world attacks [10,37].

To mitigate this concerning state of affairs for deployed software, researchers have developed techniques that identify and trigger potential vulnerabilities in software binaries without availability of source code or debug symbol information [26,6,4]. However, even when applied, these tools may not find all the vulnerabilities present in that software. Attackers can exploit these software vulnerabilities to compromise user systems and expose sensitive customer data.

Unfortunately, there is considerably little prior work on automatically identifying provisions employed by the software developer and present in the binary software to detect and prevent such attacks even when software vulnerabilities are exploited. Many such provisions to thwart software attacks are available to developers, including techniques to detect and prevent all memory attacks [21], that use stack canaries to detect some buffer overflows [7], and to prevent control-flow attacks [1], etc. To be most effective, rather than being applied manually and in an ad-hoc manner by the software developer, such security techniques must be applied automatically and systematically by a tool like the compiler during the software build process. The knowledge that the given software is protected from attacks, even in the presence of software bugs, can relieve customers and increase their confidence and comfort to use the software.

In this work we explore and develop a generalized technique to identify security checks inserted by compilers and other automated tools in binary software. Our techniques do not depend on knowledge of the source programming language, or the compiler used to insert the check, or (the signature of) the specific security check implemented. Rather, our techniques to detect security checks in binaries employ insights that we expect (and aim to verify in this work) are typical to most such checks that are inserted by automated tools, like compilers. Our experiments in this paper focus on memory-related attacks and vulnerabilities, which are dominant is binary code built from memory unsafe languages, like C/C++ [32].

Our approach employs the following unique insights to detect security checks in binaries. We hypothesize that security checks applied by automated tools will be inserted at *code sites* just before the *interesting* or vulnerable code construct they are designed to protect, like return addresses, indirect calls/jumps, and array/buffer references. We also reason that the code inserted by any specific automated security check will display a similar instruction pattern or a few sets of patterns across its multiple deployment code sites. For the class of memory-related attacks we study in this work, we also speculate that the security checks will *validate* the vulnerable memory address that must be protected. Finally, we hypothesize that, by contrast, code that is built without the security check will not typically contain any uniform code pattern across the multiple potential deployment code sites in the binary.

Our novel technique then uses these insights to identify compiler-inserted security checks in any given arbitrary binary software by following these general steps: (a) employ a reverse engineering tool (Ghidra [22], in this work) to detect the potential deployment sites in the given binary (also called *interesting* instructions or constructs in this work) for any specific security check, (b) fetch and dump disassembled code blocks around the interesting program points, (c) process the dumped instruction traces to normalize constants, register numbers, labels, etc., and (d) validate and find common instruction patterns across collected traces. Our technique uses the presence of common instruction sequences/patterns across traces to deduce the likelihood of compiler checks to protect against attacks related to that code construct.

Thus, our primary contribution in this work is the conception, development, and detailed assessment of a novel and all-inclusive technique to determine the presence of security checks inserted by automatic tools in binary software. Our work has the potential to benefit the customers and end-users of software that can now independently verify certain security aspects of the binary program without relying on information (that is mostly not) provided by the software developer.

The remainder of this paper is organized as follows. We describe background information and related works in Section 2. We present our experimental and benchmark configuration in Section 3. We explain the insights for this work in Section 4. We describe our technique in Section 5. We present our experimental results and observations in Section 6. Finally, we discuss avenues for future work and state our conclusions in Sections 7 and 8, respectively.

2 Background and Related Works

Code bugs and missing safety oversight for vulnerable code constructs are widespread [23], especially in software built using memory and type unsafe languages, like C/C++. *Memory corruption errors* have consistently ranked in the top three most dangerous software weaknesses [9]. Memory bugs can be exploited to alter the program behavior and take over program control [32], and launch many critical software attacks [36,8]. While memory-safe language alternatives are available, C/C++ remain popular [1] due to the large amount of existing legacy code, and low-level features of these languages that are desired by performance and memory critical systems.

Modern compilers provide a number of security checks to protect software and end-users from attacks that target such code vulnerabilities. Many checks are applied completely statically or at compile-time. For example, most compilers use *warnings* to indicate some potential code bugs to developers. Likewise, static analyzers can also provide algorithms to perform deeper syntactic and semantic analysis of the code to find more complex coding bugs without running the program [2]. By contrast, some security checks require run-time support and add instrumentation code in the binary to detect problems during execution. Such security checks are called *run-time checks*. In this work we focus on run-time checks that may be inserted by compile-time tools in the binary object files or executables.

For ordinary users, it is typically up to the software developers to enable (static or run-time) security checks for their software. Moreover, in most cases the software companies do not indicate whether their products were built with security checks enabled. There is also currently no way (to our knowledge) for end-users to independently determine or verify the presence of security measures in the software they own or use. Knowing this information will be useful for end-users to select the appropriate software product and use it more confidently.

[1] https://www.tiobe.com/tiobe-index/
[2] https://clang-analyzer.llvm.org/

To our knowledge, this is first work that aims to develop an automated algorithm to identify the presence of any general run-time security check in the distributed binary software. Related to our present work is prior research in binary analysis methods to address security issues, including code vulnerability detection [26,4], malware identification [35,2], and code similarity analysis.

Vulnerability detection attempts to find exploitable bugs in the binary using static binary analysis [12,14], symbolic execution [5,6], or run-time analysis [27,31] Detecting and eliminating vulnerable code from binaries will deny attackers the opportunities to compromise software, and thus protect end-user systems and data. In this respect, our work shares a similar goal with these approaches. However, our directions differ considerably. Our work is based on the assumption that vulnerability detection systems cannot detect all vulnerabilities in the binary. Indeed, none of the existing approaches claim to find all code vulnerabilities. Therefore, security checks will remain relevant, and their presence in binaries can provide end-users the assurance that their system and data will still be protected if any (remaining) vulnerabilities are exploited.

Common malware detection software, including virus scanners [3,33], employ signature based methods that identify unique strings or byte patterns in the binary code [30,15]. Signature based methods cannot protect against advanced malware that employ obfuscation, or polymorphic and metamorphic malware variants that can change their code, signatures, or other identifiable patterns to evade detection [25]. Signature based methods to detect the presence of a few known security checks in binary programs have also been developed [19], and suffer from similar issues regarding generality. To avoid a similar limitation, we do not employ signature-based methods in this work. Also different from our work, the malware detection methods rely on prior knowledge that the analyzed software is malicious and the techniques need to detect patterns that the malware is specifically attempting to hide.

Somewhat related to our current work is research in code similarity analysis that attempts to determine if distinct code regions or software are derived from the same code base [18,12,24,13]. However, research in code similarity analysis needs to address a vastly different set of challenges to identify similar codes when their representations are expected to be significantly different due to variations in, for example, hardware architectures and compiler configurations. Instead, our goal is to identify security checks inserted by automated tools that we expect will exhibit similar code patterns.

3 Evaluation Framework

In this section we describe our experimental configuration and the benchmarks used for this work. To evaluate our hypothesis and approach, we design a controlled experiment by selecting a fixed set of compiler inserted security checks in the Clang/LLVM tool-chain [28] that we can explicitly enable or disable. The first column in Table 1 lists the three security checks that we use for our experiments in this work. These are: (a) the stack canary check, *Stackguard* [20], (b)

the *Control Flow Integrity (CFI)* protection [34], and (c) a fast memory error detector, *AddressSanitizer (Adsan)* [29]. The final two columns in Figure 1 show the Clang/LLVM flags we use to explicitly enable or disable the respective security check. Additionally, all benchmarks were compiled with optimizations (-O2) ON.

Sec. Check	Enable Flag	Disable Flag
Stackguard	-fstack-protector-all	-fno-stack-protector
CFI	-flto -fsanitize=cfi fvisibility=default	-flto
Adsan	-fsanitize=address -fno-omit-frame-pointer	-fno-omit-frame-pointer

Table 1: Clang/LLVM security checks and flags to enable/disable each check

We employ twelve C/C++ programs from the SPEC cpu2006 benchmark suite for experimentation [17]. Binaries are generated for the x86-64-Linux platform. We expect our observations from this work to extend generally to other compilers, programs, and platforms.

Static analysis on the binary executables is conducted by extending the Ghidra reverse engineering framework [22]. Scripts to extend Ghidra's functionality are written in Python.

4 Insight

Code to check for program safety and security conditions may be added by the human developers or by automated tools, like the compiler. Our approach to detect security checks in binary code is guided by a few key principles and insights. We hypothesize that security checks inserted by automated algorithms follow a small number of well-defined rules with regards to when and where they are inserted and the actual instructions used at the instrumentation points, which can enable us to detect their presence in binaries. Firstly, security checks will be placed close to the actual code construct that is being protected. For example, checks to detect memory-related errors are likely to be placed just before the code that references the vulnerable memory address. Conversely, the location of the security check code can reveal the purpose of the security check. Secondly, the inserted security check code will follow a single pattern or one from a small number of code templates/patterns. Thirdly, we expect the inserted security check code to perform some operation on the protected memory address and/or test it for correctness. Finally, the inserted code will likely accomplish a task that is orthogonal or different than the primary program logic.

To verify these hypotheses we first conducted a small manual study with a few security checks inserted by standard C/C++ compilers, like GNU GCC [16] and LLVM [28], to prevent common memory corruption attacks. Figure 1 shows (portions of) the security check code inserted by three different protection mechanisms in the Clang/LLVM compiler. Figure 1(a) shows the pattern for the se-

```
mov %rax, (gbl_can_addr)
mov %rax, (%rax)
cmp %rax, (%rbp - lcl_can_off)
jnz L1
...
L1:
Call <stack_chk_fail>
```

(a) Stackguard check pattern

```
mov %rax, (ind_call_addr)
mov %rcx, chk_addr_imm
cmp %rax, %rcx
jz L1
ud2
...
L1:
call %rax
```

(b) CFI check pattern – 1

```
mov %rbx, (ind_call_addr)
mov %rax, chk_addr_imm
mov %rcx, %rbx
sub %rcx, %rax
mov %rax, %rcx
shr %rax, 0x3
shl %rcx, 0x3d
or  %rax, %rcx
cmp %rax, const_imm
jbe L1
UD2
...
L1:
call %rbx
```

(c) CFI check pattern – 2

```
mov %rax, %rdi
shr %rax, 0x3
mov %al, (%rax+const)
test %al, %al
jnz L2
L1:
reference to (%rdi)
...
L2:
mov %ecx, %edi
and %cl, 0x7
add %cl, 0x3
cmp %cl, %al
jl  L1
call <asan_chk_fail>
...
```

(d) AddressSanitizer check pattern

Fig. 1: Portions of security check code inserted by different security mechanisms built into the Clang/LLVM compiler (determined by manual code analysis)

curity check code inserted by Clang's stack canary check [20], Figure 1(b) and Figure 1(c) show two code patterns used by the control-flow integrity protection [34], and Figure 1(d) shows one of the instruction patterns used by the *AddressSanitizer* memory error detector [29].

We find that the instrumentation codes added by these three security techniques largely support our key insights that we presented earlier. By design, we do not attempt to decipher the logic of the checks, but only look for common instruction patterns. We observe specific instructions added by each security check immediately prior to the protected memory address dereference. For instance, with *Stackguard*, the instrumentation code is inserted in the (function prologue and) epilogue before the *return* instruction. With CFI, the security check is inserted before the indirect branch and indirect call instructions. All the security check codes we studied also perform a *compare-branch* to either the original (protected) program code on success, or the exception checking code on failure.

This manual study also revealed several factors the can complicate automated analysis to detect security checks patterns in binaries. Even when the instruction patterns are consistent, register and constant operands vary from one instance of the check to another. The security check is not added at every point of interest, but only when deemed necessary by the compiler to limit run-time overhead. Many security checks, including *CFI* and *AddressSanitizer*, may introduce multiple code patterns for different cases. Compiler optimizations may intersperse the instruction pattern with (unrelated) program instructions and make them harder to detect. We employ these observations to develop our algorithm to automatically detect security check code patterns in binaries.

We only perform this manual identification of security check patterns to confirm our hypothesis. Our technique described and evaluated in the remainder of the paper does not rely on any manual work, and attempts to *automatically* identify all targeted security checks and corresponding instruction patterns, when present in the binary.

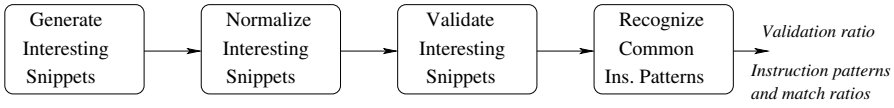


Fig. 2: Approach to detect compiler-inserted security checks in binary code

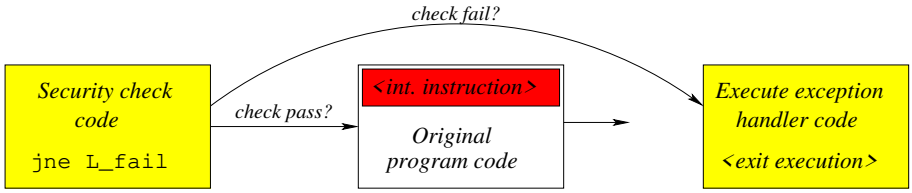


Fig. 3: Typical code structure when a security check is inserted before the interesting code construct in binaries

5 Technique to Detect Security Checks in Binaries

In this section we explain our approach to automatically detect compiler-inserted security checks in the binary code. A signature based method that *manually* collects and stores each security check pattern, and later attempts to detect the signatures in a given binary to identify enabled security checks is plausible. However, this method is tedious and intrinsically limited to only the collected instruction signatures and harder to generalize to different languages, compilers, and checks. We therefore propose a method to *automatically* determine the security checks without any a prior knowledge of known signature patterns.

Figure 2 illustrates our automatic technique to detect compiler-inserted security checks in binary object/executable files. We employ the observations and insights explained in the previous section to guide our approach. We explain each step of our technique in the remainder of this section.

5.1 Generate *Interesting* Code Snippets

The first step of our technique uses Ghidra to statically analyze the given binary and locate the memory operands, or code constructs, or instructions of interest. For the security checks against memory corruption attacks we selected for this work, our Ghidra scripts find (a) the *return* instruction for the (*stackguard*) checks to prevent control-flow hijacking by overflowing the 'return' address, (b) the *indirect branch/call* instruction for the (*CFI*) checks related to confirming control-flow integrity, and (c) any memory *address dereference* for the (*AddressSanitizer*) checks attempting to detect other memory errors, like out-of-bound memory access.

We hypothesize that the security check instructions are typically placed just before the *interesting* code and contain a compare-branch construct. The typical code structure is illustrated in Figure 3, where the nodes are basic blocks and

Algorithm 1: Validation of the code snippet

```
   Input: Vulnerable memory address
   Input: Original block containing the interesting instruction, srcBlk
 1 predBlk ← getPredecessorBlock(srcBlk) ;
 2 if predBlk == Null then  return False ;
 3 NormInsList ← Concatenate(predBlk, srcBlk) ;
 4 InsList ← Reverse(NormInsList) ;
 5 intOperList ← Vulnerable memory address ;
 6 validated ← False ;
 7 foreach ins ∈ InsList do
 8 |   if ins is Compare/test instruction then
 9 |   |   cmpOperList ← getOperands(ins) ;
10 |   |   if cmpOperList contains operand from intOperList then
11 |   |   |   validated ← True ;
12 |   |   |   return ;
13 |   |   end
14 |   |   break ;
15 |   end
16 |   else
17 |   |   dstOper ← getDestinationOperands(ins) ;
18 |   |   srcOper ← getSourceOperands(ins) ;
19 |   |   if intOperList contains operand from dstOper then
20 |   |   |   delete(intOperList, dstOper) ;
21 |   |   |   append(intOperList, srcOper) ;
22 |   |   end
23 |   end
24 end
25 foreach remaining ins ∈ InsList do
26 |   dstOper ← getDestinationOperands(ins) ;
27 |   srcOper ← getSourceOperands(ins) ;
28 |   if cmpOperList contains dstOper then
29 |   |   if intOperList contains any srcOper then
30 |   |   |   validated ← True ;
31 |   |   |   return ;
32 |   |   end
33 |   |   delete(cmpOperList, dstOper) ;
34 |   |   append(cmpOperList, srcOper) ;
35 |   end
36 end
37 return False ;
```

edges represent the control-flow between blocks. The *instruction* that is highlighted in red indicates the interesting code construct that is detected by our Ghidra based scripts for each security check. When an interesting code construct is located in the binary, we suppose the security check code to be present (at the bottom) of the *predecessor* block, and the exception code that is reached when the security check condition fails to be placed in the *other successor* block. If the security check is successful, the original program instructions are executed, but if the check fails, then the exception condition is reached.

Then, our Ghidra script concatenates the instructions in the predecessor and 'other successor' blocks to form the *interesting code snippet* for that instance of the interesting code construct. We refer to them as *'instances'*. Similarly, our analysis will find and output all the interesting code snippets in the binary for further analysis.

5.2 *Normalize* Interesting Code Snippets

Next, we generalize the instruction operands that can vary between the different occurrences of the same security check. We call this process *normalization* of the code snippets, and it is done to make it easier for automated algorithms to find common patterns in the code snippets extracted in the previous step.

We experimented with different modes of aggressiveness in this phase. The most aggressive mode eliminates all instruction operands and only keeps the opcodes in the code snippets. We settled on a less aggressive mode that maintains most of the operand context information to better balance the two objectives of reducing false positive matches while still enabling detection of similar instruction patterns. We make the following changes to the instruction operands during the normalization phase: (a) delete address offsets, (b) generalize all register names, except the stack pointer (%rsp) and the base pointer (%rbp) registers that are used to access local variables on the *stack*, (c) generalize the different JUMP statements (like JNE, JZ, JGE, etc.), (d) remove most constants and immediate operands, and (e) replace distinct label names with a constant string.

5.3 *Validate* Interesting Code Snippets

In our next optional step, we statically analyze the code snippet using another key insight of this work. Specifically, we hypothesize that, for the memory corruption attacks selected in this work, the security check codes will perform some computation on and/or comparison with the vulnerable memory address to confirm that it is not corrupted. We call this step the *validation* of the code snippet. We can see from Figure 1 that this insight is true for all the security check mechanisms employed in this work. We expect a higher likelihood of validations when the security check is enabled (in the compiler) compared to code snippets collected from binaries with the security check flags disabled.

Algorithm 1 describes the high-level steps performed during the validation process. The validation check is performed over all the code in the block containing the interesting instruction and its predecessor block. The algorithm first checks if the protected memory address or a value derived from it (dependent) is used directly in a comparison statement. If not, then the algorithm checks if the operands used in the comparison statement are derived from the protected memory address or its dependents. If true, then the algorithm declares the memory address to be *validated*.

5.4 Recognize Common Instruction Patterns

After generating, normalizing and validating the code snippets, our next task is to detect common instruction patterns over all snippets. We expect a single or a small number of dominating instruction patterns when a security check is enabled, and no single instruction pattern to dominate, otherwise. We experimented with several different pattern matching algorithms to find one that best differentiates the two categories of binaries. We describe two such algorithms in this section.

Algorithm 2: Pattern Recognition using String Compare

Input: Instances of Interesting Snippets
Input: Count of the instructions $\longrightarrow N(1 \leq N \leq 10)$

```
1  Let I_d ⟶ Set of interesting snippets;
2  Let M ⟶ Set of instruction patterns of length N;
3  Let Count ⟶ Set of count of matched patterns of length N;
4  Let LeftOut ⟶ Set of instances containing the most matched instructions of length N;
5  Let InsCount ⟶ Count of the total number of instances;
6  foreach n ∈ N (10 ⟶ 3) do
7      compare ← 0;
8      foreach i ∈ I_d do
9          foreach j ∈ I_d do
10             M_i ← last n instructions of i not in LeftOut;
11             if Last n instructions of i == Last n instructions of j then
12                 count_i ← count_i + 1
13             end
14         end
15     end
16     ratio ← count[index(maximum(count))]/InsCount;
17     if ratio > compare then
18         Final ← M.index(maximum(count));
19         M.emptyList();
20         Count.emptyList();
21         n ← n-1;
22         compare ← ratio;
23     end
24     else
25         n ← 10;
26         LeftOut ← instances whose last n instructions are identical to final;
27     end
28 end
```

Pattern Matching Using Simple String Comparison

Algorithm 2 explains our first pattern matching algorithm. Each iteration of this algorithm uses a simple string comparison over the *last 'N'* ($(3 \leq N \geq 10)$ instructions of all (unmatched) instances of interesting code snippets to find the *longest* string (sequence of instructions) that has the *highest* number of matches. The *match ratio* is computed for this string by dividing the number of code snippets it matched by the total number of interesting code snippets. Then, all the code snippets that contain this longest string are removed from consideration, and the next iteration of this algorithm is performed to find the next longest string (sequence of instructions) with the highest number of matches among the remaining code snippets. This process is repeated until the algorithm cannot find any string with at least two matches.

This algorithm is simple and fast, yet can automatically vary the pattern length ('N') across its successive iterations to find successive longest strings of matches. However, it may not work well when instruction patterns are hindered by code reordering and other compiler optimizations. Next, we describe an algorithm to overcome this limitation.

Using Longest Common Subsequence Algorithm

Our next approach uses the longest common subsequence (LCS) algorithm to find potentially non-sequential common instruction patterns across code snip-

Algorithm 3: Pattern Recognition using Longest Common subsequence

Input: Instances of Interesting Snippets

```
1  Let I_d ⟶ Set of the last 15 instructions of each instance of interesting snippets;
2  Let LCS ⟶ Set of longest common subsequences between instance i and all the other
   instances till n(i)≤ n(I_d) , where n(X)⟶ Count of instructions in X;
3  Let Y ⟶ Nested list of set of largest subsequences for each instance i;
4  Let Eqclasses ⟶ Final list of equivalence classes i;
5  foreach i ∈ I_d do
6  |   foreach j ∈ I_d do
7  |   |   LCS ← FindLongestCommonSubsequence(i,j);
8  |   end
9  |   Y ← FindSetOfLargestSubsequences(LCS);
10 end
11 foreach Y_s ∈ Y do
12 |   Eqclasses ← FindEquivanceClasses(Y_s)
13 end

14 FindEquivalenceClass(a)
15 currentString ← a;
16 foreach equalString ∈ Eqclasses do
17 |   if Levenshtein.ratio(currentString,equalString) ≤ 0.9 found FALSE then
18 |   |   Eqclasses ← currentString;
19 |   end
20 end
21 return eqclasses
```

pets. Algorithm 3 explains the main steps in this technique. The algorithm is given the set of the last *15 instructions* of all code snippets collected by Ghidra for the targeted vulnerability protection (I_d). Following our hypotheses for the *interesting snippets generation*, the security check codes, if present, should be observed at the end of the interesting code snippet.

The technique begins the first iteration by using the standard LCS algorithm to compare the first code snippet ($N = 0$) with all other code snippets ($1 \leq N \leq I_d$) to generate ($N-1$) LCS strings. It then finds the (set of) LCS string(s) with the maximum length. There could be more than one *longest* distinct LCS string. The first *longest* LCS string is put into a new *equivalence class*. Each successive longest LCS string is compared with all pre-existing equivalence classes using the Levenshtein's distance formula. [3] If the Levenshtein distance ratio is greater than 0.9, then this longest LCS string is considered to be part of the same equivalence class, and the *hit-count* of the equivalence class is incremented by one. If this longest LCS string does not match any existing equivalence class, then a new class for this LCS string is created.

Successive iterations of this algorithm repeat this process for all the other code snippets in the set. Finally, for each equivalence class, we compute the *match ratio*, which is its hit-count divided by the total hit-count over all equivalence

[3] The Levenshtein distance is a string metric for measuring the difference between two sequences. Levenshtein distance between two words is the minimum number of single-character edits required to change one word into the other. An "edit" is defined by either an insertion, a deletion, or a replacement of a character.

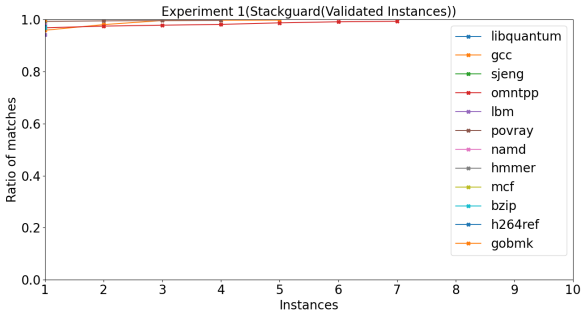


Fig. 4: Results using the simple string comparison algorithm for Stackguard.

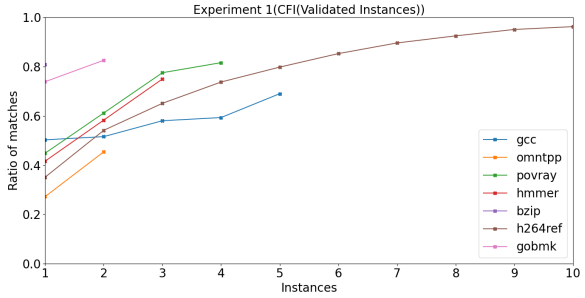


Fig. 5: Results using the simple string comparison algorithm for CFI.

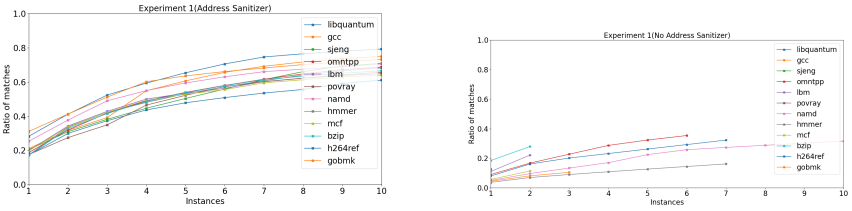
classes for that benchmark. The set of equivalence classes is sorted using this "match ratio" metric to output the set of dominating instruction patterns.

6 Experimental Results and Observations

In this section we present the results of experiments we conducted using the framework, approach and benchmarks described earlier. First, we only present the results with the two pattern matching algorithms in Sections 6.1 and 6.2, respectively. Then, we discuss our overall observations in Section 6.3.

6.1 Results with the Simple String Comparison Algorithm

We first present results with our approach using the simple string comparison algorithm for pattern matching. Figures 4 and 5 plot the sorted cumulative *match ratio* for each benchmark with the Stackguard and CFI security check enabled, respectively. In both these cases, our address validation check was very effective at eliminating almost all code snippets for most benchmarks when the respective security check was turned OFF. For the few code snippets that remained for some



(a) AddressSanitizer ON (b) AddressSanitizer OFF

Fig. 6: Results using the simple string comparison algorithm for the AddressSanitizer security check enabled and disabled.

benchmarks after validation, there were no dominant code patterns detected. Therefore, we do not plot the corresponding (blank) graphs with the Stackguard and CFI checks turned OFF.

Also, for the CFI check in Figure 5, we only plot results for seven of the eleven total benchmarks. The remaining 4 benchmarks generate very few code snippets (typically, less than 5), almost none of which are validated. Wherever fewer code constructs are generated, the curves stop after the few instances present. We manually looked at these cases and found that the compiler did not apply the CFI check for these cases even with the flag turned ON.

Figures 6(a) and 6(b) plot the sorted cumulative *match ratio* over the validated code snippets for *AddressSanitizer* with the check turned ON and OFF, respectively. We note that we have currently implemented a simplistic Ghidra-based binary analysis script for AddressSanitizer. Rather than only detecting the potentially vulnerable memory dereferences (vectors and pointers), our script is overly aggressive and currently detects all memory dereferences (even scalars) in the binary as opportunities for protection. We then rely on the validation algorithm to eliminate many of the spurious instances. Even then, we are left with a significant number of code snippets to analyze even with the AddressSanitizer security check turned OFF.

6.2 Results with the LCS Pattern Matching Algorithm

In this section we present the results of experiments that use the LCS algorithm for pattern matching. Similar to the previous section, Figures 7 and 8 plot the sorted cumulative *match ratio* for each benchmark with the Stackguard and CFI security check enabled, respectively, and using the LCS algorithm. Likewise, Figures 9(a) and 9(b) plot the sorted cumulative *match ratio* over the validated code snippets for *AddressSanitizer* with the check turned ON and OFF, respectively.

6.3 Observations

We can make a number of important observations from our experimental results. First, for the automated run-time security checks we study in this work,

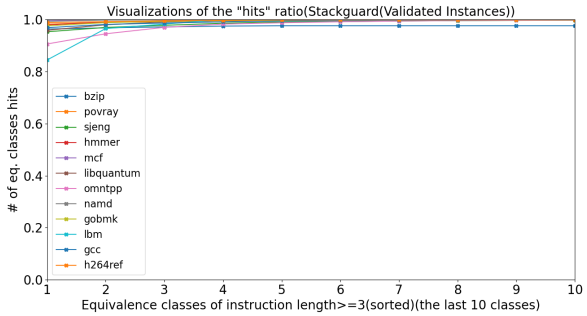


Fig. 7: Cumulative match ratio of top ten equi. classes with *Stackguard* ON

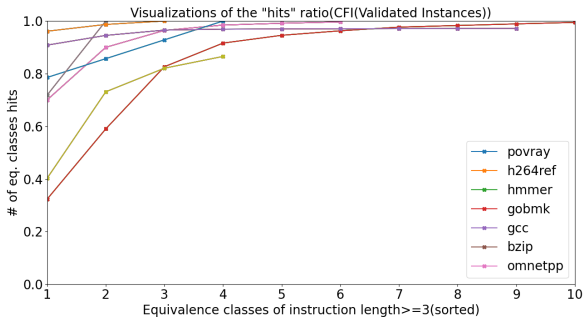
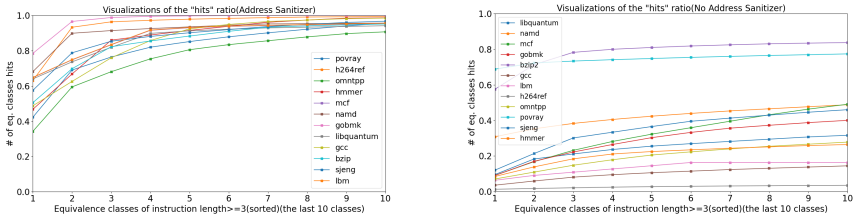


Fig. 8: Cumulative match ratio of the top ten equi. classes with *CFI* ON

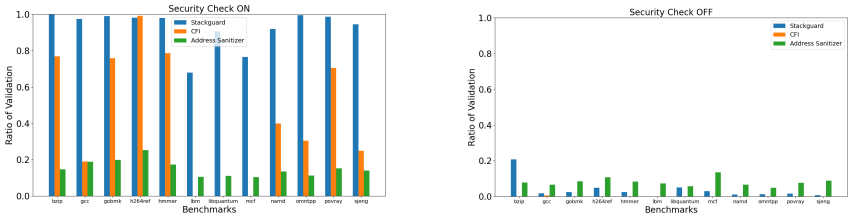
the combination of the validation and pattern matching components of our technique can clearly indicate when and which security check is enabled or disabled. We find that both our pattern matching algorithms obtain results that show a starkly different trajectory of the graph for each benchmark when the respective security check is ON, as compared to when the check is OFF. Even so, the LCS algorithm appears to be more adept at finding the common instruction patterns. Thus, while both algorithms perform well for Stackguard (Figures 4 and 7), the LCS algorithm detects pattern matches more efficiently for the CFI and AddressSanitizer checks (Figures 5 and 8, and Figures 6(a) and 9(a)).

However, the greater adeptness of LCS can also sometimes be a disadvantage. Thus, we find that the LCS algorithm produces a few false positives in the *AddressSanitizer* OFF case (Figure 9(b)) for *bzip* and *povray*). We analyzed the *bzip* benchmark and found that there is indeed a high-frequency pattern that is produced by a set of three frequently used macros. While compiler optimizations sufficiently interleave this pattern with other instructions in several cases to hide it from the simpler string comparison algorithm, the LCS algorithm finds it even when the instructions in the pattern are not consecutive.



(a) *AddressSanitizer* ON (b) *AddressSanitizer* OFF

Fig. 9: Cumulative match ratio of the top ten equivalence classes with the *AddressSanitizer* flags turned ON and OFF, respectively.



(a) Security check ON (b) Security check OFF

Fig. 10: Validation ratio of interesting code snippets when the respective security check is either ON or OFF

Second, the *validation* algorithm can effectively weed out the spurious code snippets. For stackguard and CFI OFF cases, the validation algorithm eliminates most of the code snippets (that do not contain the check) from further analysis. Figures 10(a) and (b) illustrate the validation ratio when each security check is either ON or OFF, respectively. The validation ratio is computed by dividing the number of validated code snippets by the total number of interesting code snippets generated by the Ghidra script for each security check. Assuming that our hypothesis that a security check should *test* the protected/vulnerable memory address is correct, a low validation ratio in the security check OFF case correctly indicates that no code is inserted at interesting points in the binary.

Third, the validation algorithm is consistently low in the AddressSanitizer ON case. This effect is due to the simplistic binary analysis (Ghidra) script we developed for this check, as was mentioned earlier. Improving this script to correctly locate only the vulnerable code locations is part of our future work.

Fourth, we find that the most common security patterns found by our algorithms in the security check ON cases closely match the expected patterns from the manual study in Section 4.

Fifth, we observed several cases where there are fewer interesting code constructs detected and interesting code snippets generated in the security check OFF case, compared to when the security check is ON. This discrepancy hap-

pens for two reasons. First, enabling the security check sometimes requires the compiler/linker to create or link additional code and functions in the binary. Our Ghidra script can then detect additional interesting code locations in the binary generated with the security check ON. Second, in some cases, especially when the security check is OFF, no interesting snippet may be generated at an interesting code location, if the code layout does not match that illustrated earlier in Figure 3.

7 Future Work

There are several avenues for future work. First, one limitation of our approach that is based on finding common instruction patterns over multiple code snippets is that a very small number of instances could generate results that produce misleading conclusions. Our current approach benefits from having a sizeable number of instances of each interesting code construct to detect patterns. We plan to address this limitation in future work by also developing some other indicators for detecting security checks. Second, we hypothesize that compiler-added security checks will perform tasks that are orthogonal to the primary function of the program. But, we do not yet apply and use this hypothesis. We plan to implement this measure in our future techniques. Third, we plan to improve our validation algorithm to eliminate even more spurious code snippets and improve pattern identification. Likewise, we will also develop and evaluate other pattern recognition algorithms. Finally, we would like to develop a more comprehensive metric to rate the security properties of any arbitrary binary.

8 Conclusion

Our goal in this work is to develop a generalized technique to detect compiler-based run-time security checks in any given binary. We develop several hypothesis regarding the properties of such security checks in binary code. We then devise a detailed mechanism that employs these hypothesis to achieve our goal, and conduct a comprehensive evaluation. Overall, we find that our technique is able to largely achieve our goal and in doing so, validates and confirms our hypotheses. Thus, security checks indeed appear to be consistently inserted immediately before the dereference of the protected memory address and contain a compare-branch sequence, which we use for our code snippet extraction. Likewise, we can deduce that the inserted security check code follows a small number of regular templates. Overall, our technique detects consistent instruction patterns much more regularly in the code snippets extracted when the security checks are ON, compared to when the checks are OFF. Our technique in almost all cases can correctly deduce if the respective security check is present in the binary. We expect our work will greatly benefit automated and independent security analysis of binary code for end-users when source code is unavailable.

References

1. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity principles, implementations, and applications. ACM Transactions on Information Systems Security **13**(1) (nov 2009)
2. Abijah Roseline, S., Geetha, S.: A comprehensive survey of tools and techniques mitigating computer and mobile malware attacks. Computers & Electrical Engineering **92**, 107143 (2021)
3. Aycock, J.: Computer Viruses and Malware (Advances in Information Security). Springer-Verlag, Berlin, Heidelberg (2006)
4. Brooks, T.N.: Survey of automated vulnerability detection and exploit generation techniques in cyber reasoning systems. In: Science and Information Conference. pp. 1083–1102. Springer (2018)
5. Cadar, C., Dunbar, D., Engler, D.R., et al.: Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI. vol. 8, pp. 209–224 (2008)
6. Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing mayhem on binary code. In: Proceedings of the 2012 IEEE Symposium on Security and Privacy. p. 380–394. SP '12, IEEE Computer Society, USA (2012)
7. Cowan, C., Pu, C., Maier, D., Hintony, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In: Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7. p. 5. SSYM'98 (1998)
8. CVE: A buffer overflow vulnerability in whatsapp voip stack (2019), https://www.cvedetails.com/cve/CVE-2019-3568/
9. CWE: CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html (2022)
10. Cybersecurity, U., Agency, I.S.: Top routinely exploited vulnerabilities (July 2021), https://www.cisa.gov/uscert/ncas/alerts/aa21-209a
11. Database, N.N.V.: Cvss severity distribution over time (December 2021), https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cvss-severity-distribution-over-time
12. Eschweiler, S., Yakdan, K., Gerhards-Padilla, E.: discovre: Efficient cross-architecture identification of bugs in binary code. In: NDSS. vol. 52, pp. 58–79 (2016)
13. Feng, Q., Zhou, R., Xu, C., Cheng, Y., Testa, B., Yin, H.: Scalable graph-based bug search for firmware images. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 480–491 (2016)
14. Gao, D., Reiter, M.K., Song, D.: Binhunt: Automatically finding semantic differences in binary programs. In: International Conference on Information and Communications Security. pp. 238–255. Springer (2008)
15. Griffin, K., Schneider, S., Hu, X., Chiueh, T.c.: Automatic generation of string signatures for malware detection. In: International workshop on recent advances in intrusion detection. pp. 101–120. Springer (2009)
16. Griffith, A.: GCC: The Complete Reference. McGraw-Hill, Inc., USA, 1 edn. (2002)
17. Henning, J.L.: Spec cpu2006 benchmark descriptions. SIGARCH Comput. Archit. News **34**(4), 1–17 (Sep 2006)
18. Hu, Y., Zhang, Y., Li, J., Gu, D.: Binary code clone detection across architectures and compiling configurations. In: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC). pp. 88–98 (2017)

19. Kamathe, G.: Identify security properties on linux using checksec (June 2021), https://opensource.com/article/21/6/linux-checksec
20. Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., Song, D.: Code-Pointer integrity. In: 11th USENIX Symposium on Operating Systems Design and Implementation. pp. 147–163. Broomfield, CO (Oct 2014)
21. Nagarakatte, S., Zhao, J., Martin, M.M., Zdancewic, S.: Softbound: Highly compatible and complete spatial memory safety for c. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 245–258 (2009)
22. National Security Agency ghidra, N.: Ghidra. https://www.nsa.gov/resources/everyone/ghidra/ (2019)
23. NIST: National Vulnerability Database. https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cvss-severity-distribution-over-time (2022)
24. Pewny, J., Garmany, B., Gawlik, R., Rossow, C., Holz, T.: Cross-architecture bug search in binary executables. In: 2015 IEEE Symposium on Security and Privacy. pp. 709–724. IEEE (2015)
25. Preda, M.D., Christodorescu, M., Jha, S., Debray, S.: A semantics-based approach to malware detection. ACM Trans. Program. Lang. Syst. **30**(5) (sep 2008)
26. Qasem, A., Shirani, P., Debbabi, M., Wang, L., Lebel, B., Agba, B.L.: Automatic vulnerability detection in embedded devices and firmware: Survey and layered taxonomies. ACM Comput. Surv. **54**(2) (mar 2021)
27. Rebert, A., Cha, S.K., Avgerinos, T., Foote, J., Warren, D., Grieco, G., Brumley, D.: Optimizing seed selection for fuzzing. In: 23rd USENIX Security Symposium (USENIX Security 14). pp. 861–875 (2014)
28. Sarda, S., Pandey, M.: LLVM Essentials. Packt Publishing (2015)
29. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: Addresssanitizer: A fast address sanity checker. In: USENIX ATC 2012 (2012)
30. Shabtai, A., Menahem, E., Elovici, Y.: F-sign: Automatic, function-based signature generation for malware. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) **41**(4), 494–508 (2011)
31. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution. In: NDSS. vol. 16, pp. 1–16 (2016)
32. Szekeres, L., Payer, M., Wei, T., Song, D.: Sok: Eternal war in memory. In: IEEE Symposium on Security and Privacy. p. 48–62. SP '13 (2013)
33. Szor, P.: The Art of Computer Virus Research and Defense. Addison-Wesley Professional (2005)
34. Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., Pike, G.: Enforcing Forward-Edge Control-Flow integrity in GCC & LLVM. In: 23rd USENIX Security Symposium (USENIX Security 14). pp. 941–955. USENIX Association, San Diego, CA (Aug 2014)
35. Ucci, D., Aniello, L., Baldoni, R.: Survey of machine learning techniques for malware analysis. Computers & Security **81**, 123–147 (2019)
36. Wheeler, D.A.: Preventing heartbleed. IEEE Computer **47**(8), 80–83 (2014). https://doi.org/10.1109/MC.2014.217
37. Wired: The reaper iot botnet has already infected a million networks (October 2017), https://www.wired.com/story/reaper-iot-botnet-infected-million-networks/