Precise Detection of Security Checks in Program Binaries

Koyel Pramanick and Prasad A. Kulkarni
Electrical Engineering and Computer Science, University of Kansas, Lawrence, Kansas, USA
koyel_pramanick25@ku.edu, prasadk@ku.edu

Keywords: program slicing, security checks, program binary

Abstract: *Security checks* are added to protect *vulnerable code constructs*, including certain indirect jumps and memory references, from external attacks. Detecting the presence of security checks that guard vulnerable code constructs provides an important means to evaluate the security properties of given binary software. Previous research has attempted to find such security checks guarding potential vulnerable codes in software binaries. Unfortunately, these techniques do not attempt to separate the original *program* code from the *security check* code, leading to many false positives. The security check patterns detected by such techniques are also inaccurate as they may be interspersed with program instructions. In this work, we develop a novel *program slicing* based technique to partition the original program code from any non-program instructions, including the added security checks. We define program code as instructions in the binary software that are needed to compute the original and expected program outputs. Our technique can more accurately identify the embedded security checks in program binaries with fewer false positives. Our technique can also find more precise security check code patterns in the given binary. Overall, our work can enable tools and humans to more effectively perform independent security evaluations of binary software.

1 INTRODUCTION

Software products, commercial or open-source, are typically distributed without any acknowledgment or comment about their safety and security properties. This unfortunate condition persists even as the number of reported software vulnerabilities have been increasing in number and severity for many years (Database, 2021) and software vulnerabilities have been found to cause many disastrous real-world attacks (Cybersecurity and Agency, 2021; Wired, 2017). Software is also often distributed in its binary format, which makes it even harder to independently study its security properties. We believe that an ability to independently conduct a thorough security assessment of binary code is important for the proliferation, deployment, and use of software that can ensure the safety and privacy of user systems and data.

While currently there is no known approach or framework to *measure* software security, researchers have developed client-side techniques to detect weaknesses and *vulnerabilities* in binary code (Qasem et al., 2021; Brooks, 2018). Vulnerabilities are programming bugs that can be exploited to compromise user software and systems. Vulnerability detection can be done using static analysis based techniques (Eschweiler et al., 2016; Gao et al., 2008), symbolic execution (Cadar et al., 2008; Cha et al., 2012) or by dynamic techniques, like fuzzing (Ucci et al., 2019; Abijah Roseline and Geetha, 2021). Unfortunately, none of the existing vulnerability detection techniques can ensure the detection and elimination of all program vulnerabilities for binary code.

Researchers have also developed mechanisms to extract *intrinsic* development-time properties of the software from the distributed binary code. Such techniques could be used to evaluate software security with the hypothesis that well-written software may be more resistant to attacks. Specifically, software that is written using safe high-level languages, using secure coding standards and software engineering principles (Howard and Lipner, 2006), and hardened with appropriate build-time compiler flags (OpenSSF, 2024) may be more protected against external attacks. For instance, a machine learning based technique was devised to determine the high-level source programming language used for coding any given binary software (Adhikari and Kulkarni, 2022). Such information is a useful metric to assess a software's security properties as *memory errors* in low-level languages, like C/C++, are known to cause numerous memory corruption errors (NIST, 2022), software vul-

nerabilities (Szekeres et al., 2013) and software attacks (Wheeler, 2014; CVE, 2019).

Likewise, researchers recently proposed a new technique to detect the presence of *run-time security checks* in software binaries (Pramanick and Kulkarni, 2022). Such checks to guard vulnerable code constructs may be inserted by developers during coding or by tools like compilers during code generation. Run-time security checks are especially important as they can ensure software security even when any existing vulnerabilities are exploited, and therefore present an attractive approach to evaluate intrinsic software security. Thus, accurately detecting the checks guarding vulnerable code constructs can provide a crucial indicator of software security.

Unfortunately, we found certain limitations in the implementation of this earlier technique that can cause many false positives in certain situations. A high false positive rate in this technique makes it hard to ascertain if the targeted vulnerable code fragments in a binary are adequately protected by run-time security checks, and may affect the usefulness of the technique. The current technique can be easily tricked by compiler optimizations and obfuscation techniques by contaminating the security code fragments by intermixing the *uniform* security check code with *random* other program instructions. It is hard in such cases for this technique to correctly identify the actual security check code present in the program binary.

In this work, we propose a *program slicing* based approach to resolve this limitation. Program slicing is a common compiler technique that computes the subset of program instructions needed to affect the values at some point of interest, specified by the user (Weiser, 1981). This program subset is called the program slice. In this work, we use slicing to partition the binary code into two sets, *program* and *non-program* instructions. The program slice only includes instructions that are necessary to compute the original program state, as written by the software developer. The added security checks will be part of the non-program instructions.

To identify the program instructions in every function, we suppose that program instructions are those that compute *state* that escapes from the function. We further suppose that state or values generated in a function can escape or leave the function through, (a) the return value, (b) arguments leaving via `call` instructions in the function, (c) writes to non-stack (global and heap) memory, and (d) writes to arguments passed by reference. We develop an approach that performs slicing over this set of instructions to identify all other program instructions that compute and facilitate state to escape from the function.

The integration of this program slicing based extension into the original technique enables it to significantly reduce false positives, and prune and more accurately identify the actual security check code that may be added to the program binary. Thus, we make the following contributions in this work.

- We illustrate the problems in the earlier technique used to identify and detect runtime security checks in program binaries.
- We develop a novel program slicing based approach to partition the binary code into *program* and *non-program* instructions, and use this partition to more precisely detect the inserted security checks, and
- We implement our technique in a state-of-the-art binary analyzer, and conduct a thorough evaluation of its properties and performance.

In the remainder of the paper, we present relevant details regarding the original technique to detect security checks in program binaries in Section 2. We present our novel slicing-based algorithm to partition the program in program and non-program code in Section 3. We present our experimental framework in Section 4. We explain our results in Section 5. Finally, we present our conclusions in Section 6.

2 Background

In this work we build on the security-check detection framework proposed by (Pramanick and Kulkarni, 2022). This earlier framework provides a novel theory and robust implementation to detect security checks in unknown program binaries.

However, this framework still suffers from a high false positive rate in some cases, and is unable to detect the precise security check code inserted into the given binary. In this section we explain the insights used in this earlier work and describe their basic technique to provide a foundation to understanding the extension that we propose and implement in this work.

2.1 Insights Used

The prior work posits that security checks in binary code are positioned near the specific code constructs they aim to safeguard, which aids in their detection. It further observes that these security checks typically follow a consistent pattern: code that inspects a specific aspect of the program is followed by a pass/fail decision. A pass decision allows the program to continue executing the protected construct, while a fail decision invokes an exception routine.

The framework validates the extracted snippets by confirming that they perform operations on the memory address or code construct they aim to protect. To generalize the detection across diverse implementations, it employs a longest common subsequence algorithm to identify recurring patterns among validated security check instances. The authors find that binaries with compiler-inserted security checks often exhibit a small and consistent set of patterns. However, the recognition of these patterns is hindered by noise introduced by *program* instructions surrounding the security checks, limiting the framework's ability to extract precise patterns.

2.2 Methodology

The prior work proposed a method to detect compiler-inserted security checks in binary code, addressing the limitations of signature-based approaches. Signature-based methods rely on manually identifying specific instruction patterns, which is labor-intensive and fails to generalize across compilers, programming languages, and types of security mechanisms. This section summarizes their methodology.

Figure 1 provides an overview of the methodology, illustrating its key steps. The framework begins with the static analysis of binary code using Ghidra (National Security Agency, 2019). This analysis identifies "interesting code snippets" associated with potential security checks, such as stack canary instructions for mitigating stack overflows, indirect branches for Control-Flow Integrity (CFI) validation, and memory references flagged by AddressSanitizer for detecting memory errors. These snippets are extracted as candidates for further analysis.

Next, the framework *validates* the snippets to ensure they correspond to security checks. This validation is guided by the hypothesis that security check instructions operate on or verify values derived from vulnerable memory addresses. By examining the relationship between these instructions and the relevant memory addresses, only those snippets meeting the validation criteria are retained.

The validated snippets are then *normalized* to address minor structural variations while preserving essential contextual information. These normalized snippets are analyzed to detect recurring instruction patterns indicative of security checks. By grouping these patterns into equivalence classes, the framework identifies common sequences across diverse implementations, even when compiler optimizations introduce variations.

3 Eliminate program instructions using *slicing*

While the earlier methodology explained in the preceding section provides a robust approach, it is limited by the presence of noise from program instructions surrounding the security checks, which hinders the recognition of precise patterns. To address this limitation, we introduce a slicing step as an extension to the framework. This slicing step systematically removes instructions that contribute to the program's primary functionality, isolating the security check instructions. By reducing noise, the slicing approach enhances the framework's ability to identify and analyze security check patterns. Details of the slicing process, its implementation, and its impact are explained in this section.

Program slicing is a widely used technique in software analysis that isolates portions of code relevant to a specific computation or aspect of interest. It is particularly useful in debugging, testing, and program comprehension. Techniques such as static slicing, which analyzes the program without executing it, and dynamic slicing, which considers specific program executions, are well-established in the field (De Lucia, 2001). Extensions to traditional slicing methods, such as symbolic slicing for enhanced efficiency (Zhang, 2019) or handling constructs like unconditional jumps (Galindo et al., 2022), have expanded its applicability. These methods typically rely on data dependency and control flow analysis to extract meaningful slices.

In this work we employ a comprehensive *slicing* step to improve the detection of security checks in binary code. Our technique focuses on isolating security check instructions from the surrounding program logic in disassembled binary code. By systematically removing instructions related to the program's primary functionality, our slicing approach reduces noise in the analysis, enabling more precise validation and pattern recognition of security checks.

Even when a security check is present, the code snippet extracted by the earlier technique may contain additional *program instructions*. We define "program instructions" as those essential for generating the expected program output or results. The inclusion of program instructions in extracted snippets introduces noise, making it difficult to identify common patterns for the security checks across multiple snippets in the binary. Our goal is to remove program instructions from the code snippet. Ideally, this process will leave us with a snippet that contains only the security check instructions when the check is present, and an empty snippet when no security check is inserted.

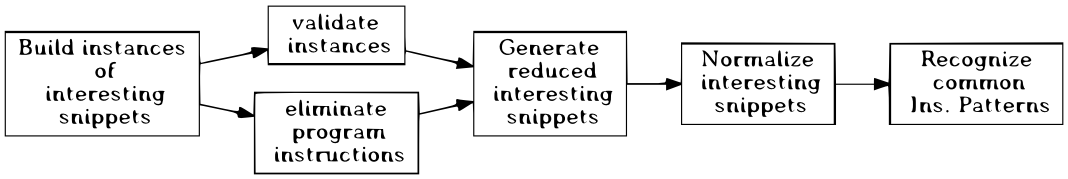


Figure 1: Strategy for identifying security checks inserted by the compiler in binary code

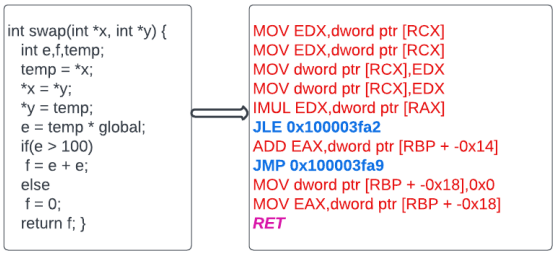


Figure 2: Code Snippet to illustrate slicing

To identify program instructions in every function, we suppose that these instructions compute *state* that escapes from the function. We further suppose that state or values generated in a function can escape or leave the function through, (a) the return value, (b) arguments leaving via `call` instructions in the function, and (c) writes to non-stack (global and heap) memory and to arguments passed by reference. Then, we develop an approach that performs *slicing* over this set of instructions to identify all other program instructions that compute and facilitate state to escape from the function. A *program slice* represents a semantically meaningful subset of computations within a program. As a decomposition technique, slicing isolates the specific computation of interest by removing unrelated program components, thereby improving the relevance and focus of the remaining code.

We explain our slicing based technique to identify program instructions in this section. We extend the Ghidra reverse engineering framework (National Security Agency ghidra, 2019) to implement our algorithm. Algorithm 1 describes the steps of our slicing algorithm. We use the code example in Figure 2 to illustrate the steps of our slicing algorithm.

3.1 Slicing Algorithm: *Step 1*

In this initial step, we employ Ghidra's slicing API to perform slicing based on three categories of instructions, as identified earlier. Thus, we slice on the arguments of the `return` instruction, `call` instruction, and in `store` instructions that write to the memory not on the current function's stack. We develop additional heuristics to address certain challenging conditions. Firstly, even security checks use function calls in exceptional situations, such as when the check fails to display the error message and exit the program. Slicing based on these function calls may incorrectly classify security check instructions as program instructions. Therefore, we employ Ghidra's API to detect such *non-returning* functions and avoid slicing on any 'calls' to non-returning functions.[1] Secondly, only store instructions that write to memory beyond the current function's stack space can lead to computation escaping the current function. We identify such store instructions by assuming that all memory references to the current stack are made using offsets from the stack/frame pointer. The remaining store instructions serve as starting points for slicing. While these heuristics are effective, they may introduce inaccuracies in categorizing sets of program and non-program instructions. For the example program in Figure 2, the assembly lines shown in red indicate the instructions detected as program instructions by this step.

3.2 Slicing Optimization 1: *Step 2*

Ghidra's slicing algorithm is implemented in their *decompiler* framework and works on a machine-independent representation called *P-Codes*. The P-Code representation of a binary in Ghidra is higher-level than the disassembly representation, and eliminates binary-level instructions that perform such tasks as managing the calling conventions during function entry/exit and calls. We consider such instructions as part of the program instructions. In this step we eliminate instructions that are typically used by the calling convention to transfer function arguments into registers. Additionally, we remove instructions related to stack management, such as the PUSH and POP instructions at the start of the function, and the

[1] Although most exception functions employed by security checks are non-returning, it's worth noting that certain exception functions utilized by the *Address Sanitizer* check don't exit directly. Instead, they invoke other functions that, in turn, contain the exit statement. These *indirect* non-returning functions pose a challenge for automatic detection by Ghidra. While ongoing efforts are directed toward refining our automatic detection approach for such functions, it's important to mention that, for the purposes of this study, we manually classified them as non-returning.

function epilogue just before the RETURN statement, which are responsible for allocating and releasing stack space. For the example program in Figure 2, the italized assembly lines shown in pink indicate the additional instructions detected as program instructions by this step in the algorithm.

3.3 Slicing Optimization 2: *Step 3*

Ghidra's slicing algorithm, even after our Step-2, leaves a substantial number of assembly instructions in the program as unclassified. In this step, we implement a simple iterative data-flow algorithm to find such additional program instructions that have a dependency on the instructions already in the current program slice. Thus, within the slice list, we track the source registers of each instruction and append to the list the most recent instructions within the block or its source block that update these registers. This approach enables us to find a larger portion of the program's instructions that were missed by the original slicing algorithm. We also handle some branch instructions in this step. If a conditional or unconditional jump statement leads to a target basic block(s) where some instructions in each of the successor basic blocks are part of the slice, then we add the compare-branch instructions to the program slice. If a block ending with a conditional branch or jump has a successor block where no instructions are part of the program slice, then those compare-branch instructions are not added to the slice.

For the example program in Figure 2, the assembly lines shown in blue indicate the additional instructions detected as program instructions by this step in the algorithm. At the end of this step, we eliminate the *program instructions* from each code snippet. This step produces a smaller and more refined code snippet. Even after completing all the algorithmic steps, certain program instructions persist due to inaccuracies in the slicing process performed by Ghidra. These remnants are essentially treated as false positives. The instructions that remain may also suggest the existence of another security check added either by the compiler or the developer.

Following the slicing process, the snippets are normalized to address minor variations in their structure while preserving essential contextual information. These refined and normalized snippets are then analyzed to identify recurring instruction patterns indicative of security checks. By grouping these patterns into equivalence classes, the framework captures dominant sequences, even when compiler optimizations introduce variations in instruction placement. This final step confirms the presence of security checks in the binary code.

```
Input: function,Function
instructionList ← getHighFunctionInstructions(Function) ;
Let STORE ⟶ Store instruction not on stack;
Let back_slice ⟶ Set of backward slices from GHIDRA;
Let step1 ⟶ Set of program instructions part of the slice after
GHIDRA's slicing;
Let step2 ⟶ Set of program instructions part of the slice after
adding instructions part of the calling convention;
Let step3 ⟶ Set of program instructions part of the slice after
adding instructions following the control flow and data flow
within the program;
foreach instruction ∈ instructionList do
    mnemonic ← getMnemonicString(instruction) ;
    if mnemonic is "CALL"/"RETURN"/STORE then
        argList ← getArguments(instruction) ;
        foreach arg ∈ argList do
            bws ← getBackwardSlice(arg) ;
            if bws is not NULL then
                backslice ← bws ;
            end
        end
    end
end
step1 ← (instructionList − back_slice) ;
foreach instruction ∈ step1 do
    mnemonic ← getMnemonicString(instruction) ;
    if mnemonic is "CALL" then
        step1 ← instructions updating registers used to set up
          function arguments ;
    end
    if mnemonic is "RETURN" then
        step1 ← instructions updating the RSP and RAX
          registers ;
    end
end
step2 ← (step1) ;
foreach instruction ∈ instructionList do
    mnemonic ← getMnemonicString(instruction) ;
    if mnemonic startswith "J" then
        srcblk ← getSourceBlock(instruction) ;
        predblk ← getPredecessorBlock(srcblk) ;
        if getInstructionsIn(predblk) in step2 then
            step2 ← instruction ;
        end
    end
end
foreach instruction ∈ instructionList do
    oper ← getOperandsAt(instruction) ;
    srcblk ← getSourceBlock(instruction) ;
    predblk ← getPredecessorBlock(srcblk) ;
    if oper is set in predblk then
        step2 ← instruction "set"ing oper ;
    end
    oper is set in srcblk step2 ← instruction "set"ing oper ;
end
step3 ← (step2) ;
return step3 ;
```

Algorithm 1: Elimination of program instructions by slicing

4 Experimental Framework

We designed a controlled experimental setup to evaluate and study the impact of our slicing-based algorithm for more accurately detecting security checks in software binaries. This setup employs two benchmark configurations:

(a) **Set-A**: All benchmarks in this configuration are compiled with one security check explicitly and intentionally enabled.

(b) **Set-B**: Benchmarks in this configuration are compiled with all security check flags disabled.

Programs in the **Set-A** configuration are further classified according to the security check that is enabled into three classes: Set-A-Stackguard, Set-A-CFI, and Set-A-AddressSanitizer, where only the compiler flags to enable the *Stackguard* (Kuznetsov et al., 2014), *control-flow-integrity* (Tice et al., 2014) and *address-sanitizer* (Serebryany et al., 2012) security checks are turned ON, respectively. Note that even when we disable the security flags in the Set-B configuration, the compiler may still add some important checks by default and the binary can still contain security check instructions added by the programmer. We design experiments to confirm our hypotheses regarding compiler-inserted security checks, and study if our approach can correctly and consistently identify the security checks in Set-A, while not producing any false positives for Set-B benchmarks.

Table 1 enumerates the configurations used for Clang/LLVM tool-chain (Sarda and Pandey, 2015) and the GCC compiler to generate our Set-A binaries with different security checks. The first column in the table lists the three checks that we employ for the evaluation of our approach in this work. The last column displays the flags used to enable or disable the respective security checks. All benchmarks were compiled with optimizations (-O2) enabled.

We utilize twelve C/C++ programs sourced from the SPEC cpu2006 benchmark suite for our experiments (Henning, 2006). The binaries are produced through the compilation of C/C++ programs using designated Clang/LLVM and GCC compiler flags tailored for the x86-64-Linux platform. We believe that the selected benchmarks, compilers and configuration options provide a sufficient platform to validate and analyze our hypotheses. Moreover, we expect that the insights and observations from this study will apply broadly to other programs, and compiler and hardware configurations. We developed scripts in Python to significantly extend Ghidra's (National Security Agency ghidra, 2019) slicing functionality and to collect information from the binaries.

Table 1: The configuration used for enabling/disabling the respective security check in CLANG and GCC

Sec. Ck.	Compiler	Check	Flags
Stackguard	CLANG	ON	-fstack-protector-all
		OFF	-fno-stack-protector
	GCC	ON	-fstack-protector-all
		OFF	-fno-stack-protector
Adsan	CLANG	ON	-fsanitize=address -fno-omit-frame-pointer
		OFF	-fno-omit-frame-pointer
	GCC	ON	-fsanitize=address -fno-omit-frame-pointer
		OFF	-fno-omit-frame-pointer
CFI	CLANG	ON	-flto -fsanitize=cfi fvisibility=default
		OFF	-flto
	GCC	ON	-fcf-protection=full -fno-sanitize=all
		OFF	fno-sanitize=all

5 Results and Observations

In this section, we present the results of our experiments and explain the benefits of our slicing-based extension to more accurately detect security checks in program binaries. First, we present results demonstrating improvements from our technique to the number and quality of snippets that are generated for each benchmark configuration. We then show the benefits of our approach to detect any common instruction patterns across the multiple code snippets for each benchmark-configuration and compiler.

5.1 Effect on Generated Code Snippets

In this section we present results about the number of collected code snippets for each benchmark configuration described in Section 4. We also present and discuss the effectiveness of our methods to prune the number of false snippets and instructions from the snippets. We only present the results obtained for binaries compiled by the Clang compiler here. Results with the GCC compiler reveal identical trends and are left out due to space considerations.

Stackguard: Tables 2 and 3 present the results for the Stackguard security check with the Clang compiler, for two benchmark configurations with the Stackguard check turned ON and OFF, respectively. The first column lists the benchmark name, followed by the number of code snippets (or *indicators*) in the next column, representing the vulnerable code sites. The following four pairs of columns present the number of valid code snippets and the average number of instructions per snippet after validation and following the application of the slicing techniques described in Sections 3.1, 3.2, and 3.3, respectively.

Table 2: Results with the Set-A-Stackguard (Check ON) configuration (with CLANG)

Benchmarks	No. of indicators	Total No. of validations							
		No Slicing		STEP 1		Step 2		Step 3	
		Valid Snips.	Avg. ins.	Valid Snips.	Avg. ins.	Valid Snips.	Avg. ins.	Valid Snips.	Avg. ins.
Bzip	61	61	8	61	4	61	4	61	4
Gcc	3377	3293	12	3221	4	3217	4	3217	4
Gobmk	2523	2496	9	2496	4	2496	4	2496	4
H264ref	538	528	10	527	4	527	4	524	4
Hmmer	494	483	7	483	4	483	4	483	4
Lbm	26	17	9	17	5	17	5	17	5
Libquantum	107	96	7	96	4	96	4	96	4
Mcf	35	26	5	26	3	26	3	26	3
Namd	100	91	8	91	4	91	4	91	4
Omnetpp	2000	1988	8	1988	4	1988	4	1988	4
Povray	1591	1573	11	1569	4	1569	4	1567	4
Sjeng	148	139	9	139	5	139	5	139	4
AVG	916.7	899	8.5	892.833	4.1	892.5	4.09	892.083	4

Table 3: Results for the Set-B configuration (with CLANG), assessed for Stackguard

Benchmarks	No. of indicators	Total No. of validations							
		No Slicing		STEP 1		Step 2		Step 3	
		Valid Snips.	Avg. ins.	Valid Snips.	Avg. ins.	Valid Snips.	Avg. ins.	Valid Snips.	Avg. ins.
Bzip	64	12	5	11	4	11	4	11	4
Gcc	3151	71	13	37	4	37	4	23	4
Gobmk	2286	44	13	24	5	23	5	7	5
H264ref	514	28	24	23	6	23	6	13	6
Hmmer	454	14	15	8	6	8	6	5	5
Lbm	21	0	0	0	0	0	0	0	0
Libquantum	101	4	14	2	4	3	4	0	0
Mcf	35	0	0	0	0	0	0	0	0
Namd	100	4	67	4	14	4	14	4	12
Omnetpp	1444	21	16	12	4	12	4	5	4
Povray	1424	43	37	24	6	23	6	10	9
Sjeng	154	3	28	3	12	3	12	3	10
AVG	812.3	20.3	19.3	12.333	5.4	12.25	5.4	6.75	4.92

We can see that the number of indicators is similar in both configurations, which is expected as there are a similar number of `return` instructions in the corresponding benchmarks in each case. However, we find that while most code snippets pass the validation algorithm for the Set-A-Stackguard configuration, a very small portion of the snippets do so for the Set-B configuration. This result indicates the importance of the *validation* step in the original algorithm to prune the spurious code snippets (that do not contain the security check in Set-B), while retaining the snippets with potential security check instructions in Set-A.

In table 2, there is little change in the number of validated code snippets at various slicing stages, showing a smaller impact of my slicing-based extension for this scenario. However, there is a substantial 53% reduction in the average number of instructions, as the slicing algorithm removes many non-security check-related program instructions from the code snippets.

By contrast, Table 3 shows a significant 67% reduction in validated code segments and a 75% reduction in the average number of instructions after applying all stages of the slicing algorithm. The slicing algorithm effectively removes most instructions, leading to a notable reduction in both these metrics. The remaining instructions and fragments are potentially *false positives* from our slicing algorithm. These false positives may consist of program instructions mistakenly identified as non-program instructions due to the conservative nature of our slicing algorithm implementation used to detect *program* instructions. In some instances, they may include security check instructions inserted by the compiler at specific vulnerable sites, even when we disable the security checks. The false positive instructions may also include other

Table 4: Results for Set-A-CFI (Check ON) configuration (with CLANG)

Benchmarks	No. of indicators	Total No. of validations							
		No Slicing		STEP 1		Step 2		Step 3	
		Valid Snips.	Avg. ins.	Valid Snips.	Avg. ins.	Valid Snips.	Avg. ins.	Valid Snips.	Avg. ins.
Bzip	24	20	7	20	4	20	4	20	4
Gcc	343	154	12	77	6	77	6	77	6
Gobmk	29	22	7	22	5	22	5	22	5
H264ref	349	346	14	346	4	346	4	346	4
Hmmer	14	11	6	10	3	10	3	10	3
Lbm	1	0	0	0	0	0	0	0	0
Libquantum	1	0	0	0	0	0	0	0	0
Mcf	1	0	0	0	0	0	0	0	0
Namd	5	2	7	2	3	2	3	2	3
Omnetpp	69	21	6	21	4	21	4	21	4
Povray	68	48	11	48	6	48	6	48	6
Sjeng	4	1	12	1	10	1	10	1	8
AVG	75.7	52.1	6.8	45.6	3.8	45.6	3.8	45.6	3.6

Table 5: Results for Set-B configuration (with CLANG), assessed for CFI

Benchmarks	No. of indicators	Total No. of validations							
		No Slicing		STEP 1		Step 2		Step 3	
		Valid Snips.	Avg. ins.	Valid Snips.	Avg. ins.	Valid Snips.	Avg. ins.	Valid Snips.	Avg. ins.
Bzip	23	0	0	0	0	0	0	0	0
Gcc	383	4	9	3	3	3	3	2	4
Gobmk	14	0	0	0	0	0	0	0	0
H264ref	3	0	0	0	0	0	0	0	0
Hmmer	4	0	0	0	0	0	0	0	0
Lbm	1	0	0	0	0	0	0	0	0
Libquantum	1	0	0	0	0	0	0	0	0
Mcf	1	0	0	0	0	0	0	0	0
Namd	1	0	0	0	0	0	0	0	0
Omnetpp	149	0	0	0	0	0	0	0	0
Povray	27	0	0	0	0	0	0	0	0
Sjeng	3	0	0	0	0	0	0	0	0
AVG	50.75	4	9	3	3	3	3	2	4

unrelated *non-program* instructions in the binary. We plan to conduct a more thorough analysis and resolution of these false positive instructions in future work.

CFI: Table 4 and 5 present our findings when the CFI check in the CLANG compiler is enabled and disabled, respectively. It's worth noting that CFI doesn't find opportunities to insert any security check instrumentation for several benchmarks that do not contain any indirect calls, particularly the smaller ones, like *Lbm*, *Mcf*, and *Libquantum*.

Surprisingly, we find that the number of indicators (indirect branches/calls, in this case) and corresponding number of code snippets vary for a few benchmarks when compiled for the SET-A-CFI and SET-B-CFI configurations, such as *Gobmk* and *H264ref*. We have not yet analyzed the reasons for this disparity in code generated by the compiler in these two cases.

Similar to the SET-A-Stackguard case, we find that in the SET-A-CFI case, most code snippets are validated, and potentially contain a security check. We again find that the validation algorithm effectively eliminates code snippets where the security check isn't present (SET-B configuration). After validation, most of the rows in Table 5 are blank.

We observe that our slicing algorithm effectively removes many program instructions within the security check snippets. This benefit is evident from the substantial 47% reduction in the average number of instructions per snippet, as shown in Table 4, when comparing the results before slicing to those after the final slicing step is applied.

Our slicing algorithm also shows notable improvements in the SET-B-CFI cases, as highlighted in Table 5. The reduction in the average number of instructions is even greater in the SET-B-CFI cases, reaching 55%. This is expected, as the SET-B configuration

Table 6: Results for Set-A-AddressSanitizer (Check ON) (with CLANG)

Benchmarks	No. of indicators	Total No. of validations							
		No Slicing		STEP 1		Step 2		Step 3	
		Valid Snips.	Avg. ins.	Valid Snips.	Avg. ins.	Valid Snips.	Avg. ins.	Valid Snips.	Avg. ins.
Bzip	7814	3993	9	2925	5	2926	5	2806	4
Gcc	163550	59844	9	57322	5	57257	5	56992	5
Gobmk	57630	25308	10	16092	5	16091	5	15999	5
H264ref	65890	40114	9	29762	5	29762	5	29608	5
Hmmer	33256	12322	9	11245	5	11242	5	11225	5
Lbm	915	36	9	32	5	32	5	30	5
Libquantum	2421	613	9	548	5	548	5	503	5
Mcf	1449	80	10	74	5	74	5	68	5
Namd	25125	5451	10	5372	5	5372	5	5321	5
Omnetpp	26097	6912	10	5019	5	5020	5	5007	5
Povray	96071	22688	10	20413	5	20408	5	20365	5
Sjeng	10274	4351	9	4156	5	4154	5	4122	5
AVG	40874.3	15142.7	9.3	12746.67	5	12740.5	5	12670.5	5

Table 7: Results for Set-B (with CLANG), assessed for Address Sanitizer

Benchmarks	No. of indicators	Total No. of validations							
		No Slicing		STEP 1		Step 2		Step 3	
		Valid Snips.	Avg. ins.	Valid Snips.	Avg. ins.	Valid Snips.	Avg. ins.	Valid Snips.	Avg. ins.
Bzip	4541	167	19	92	4	92	4	24	7
Gcc	79292	3104	21	1469	4	1462	4	803	4
Gobmk	21137	436	14	282	5	282	5	171	4
H264ref	33467	708	32	434	8	427	7	281	8
Hmmer	14117	369	17	262	5	258	5	120	5
Lbm	390	7	69	7	4	7	4	4	2
Libquantum	747	29	23	21	6	21	5	17	5
Mcf	637	16	18	9	4	8	5	5	3
Namd	10278	163	12	135	4	135	4	128	3
Omnetpp	13458	318	29	173	4	165	4	76	3
Povray	41271	644	20	426	3	405	3	137	4
Sjeng	4974	108	17	65	4	62	4	45	3
AVG	18692.4	57205.8	24.3	281.25	4.58	277	4.5	150.92	4.3

may lack true security checks, allowing our algorithm to significantly reduce false positives in the original algorithm and improve the accuracy of the results.

Address Sanitizer: Tables 6 and 7 present the results obtained with the SET-A-AddressSanitizer (check ON) configuration and SET-B configuration (check OFF) for the Clang compiler. For this work, we wrote a simple Ghidra-based binary analysis script to identify the *indicators* for the Address Sanitizer check. Our improved script builds upon the original work by incorporating type analysis to differentiate vector memory accesses from scalar accesses. Unlike the original approach, which treats all memory dereferences as potential areas for protection, our script identifies vector memory accesses separately, providing a more targeted and effective approach. Interestingly, we find that the improved validation technique and new slicing algorithms are highly effective at eliminating the numerous spurious instances.

In Table 6, we observe a 16% reduction in the number of code snippets after the final stage of slicing compared to the scenario with no slicing. This reduction highlights the effectiveness of the slicing process in eliminating spurious code snippets that were mistakenly identified as security check instructions. The initial slicing step (*STEP 1*) is particularly adept at identifying such code fragments, resulting in a relatively smaller decrease in the subsequent steps (*STEP 2* and *STEP 3*). Furthermore, there is a significant 46% decrease in the average number of instructions within the remaining code snippets, showing effectiveness at eliminating program instructions from the extracted snippets.

In Table 7, a remarkable 70% reduction is observed in the number of verified code constructs after the final stage of slicing, compared to the scenario

with no slicing. Notably, when the security check is turned off, a significant decrease in the number of validations is evident during the first and third stages of slicing, amounting to a 46% reduction, which underscores the accuracy of our algorithm in each step. Additionally, there is a substantial 82% decrease in the average number of instructions. This reduction can be attributed to the slicing algorithm's capability to isolate security check instructions by filtering out most program-related instructions.

5.2 Common Instruction Patterns

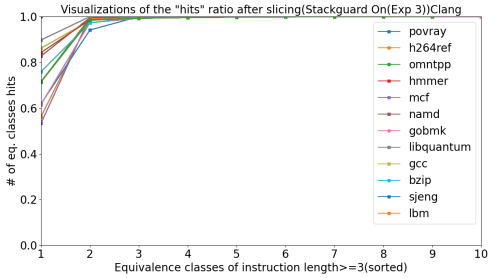
The final step in the original algorithm is to detect common instruction patterns among the extracted and validated snippets for each configuration. The Longest Common Subsequence algorithm is used for the pattern matching. The pattern matching algorithm counts the number of *hits* to each unique instruction pattern (or, equivalence class) for the code snippets for each benchmark. The instruction patterns are then sorted by their number of hits. The sorted fraction of hits, also called the *match ratio* is plotted in the figures in this section.

With a precise implementation of the proposed technique, the hypothesis predicts the SET-A configurations (security check ON) to deliver just one or a few high-frequency patterns corresponding to the actual compiler-inserted security check(s). In contrast, the hypothesis predicts SET-B configurations to not reveal any high-frequency instruction pattern since the check is turned off, and the validated snippets likely only contain *false positive* program instructions left behind by our conservative slicing algorithm.

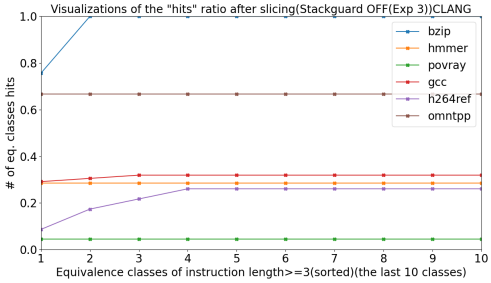
We present our observations from this pattern recognition step in this section. Again, to conserve space, we only present results obtained from binaries compiled using the Clang compiler. Results with the GCC-compiled binaries are not included, but are comparable to those presented here.

5.2.1 Security Check Pattern for Stackguard

Figures 3(a) and 3(b) display the sorted cumulative *match ratios* for each benchmark for the Stackguard security check with CLANG for the SET-A-Stackguard and SET-B configurations, respectively. As expected, the algorithm finds that a single dominant instruction pattern accounts for a large majority of hits in the SET-A-Stackguard configuration (Figure 3(a)). In fact, just two instruction patterns (that are small variants of each other) are present in all the extracted and cleaned instruction patterns in each benchmark. Furthermore, the same high-frequency instruction patterns are found across all the benchmarks.



(a) Stack Protection ON (SET-A)



(b) Stack Protection OFF (SET-B)

Figure 3: Pattern recognition results for Stackguard

In contrast, we do not find one dominant instruction pattern among the code snippets for most benchmarks in the SET-B configuration for Stackguard (Figure 3(b)) Interestingly, our algorithm identifies a high-frequency pattern in the SET-B configuration with *Stackguard* (check OFF) for *bzip*. Upon manual inspection, we discovered that, despite turning off the security check, the compiler introduces the stackguard check in a few locations of the program binary. It is encouraging to note that the algorithm successfully detects the presence of this pattern in the binary.

We also analyzed the high fraction hit count for some patterns in the other benchmarks in Figure 3(b). We attribute this seeming anomaly to the small number of validated code snippets that remained after the elimination steps and small patterns.

5.2.2 Security Check Pattern for CFI

Figure 4 presents the pattern recognition results for just the SET-A-CFI configuration with the Clang compiler for only seven out of our twelve total benchmarks. The remaining five benchmarks (*lbm*, *libquantum*, *mcf*, *namd*, and *sjeng*) yield very few code snippets, typically less than five, and almost none of them pass the validation process, as seen from Table 4. Upon manual inspection, we found that the compiler did not apply the CFI check for some of these cases even with the flag turned ON. Again we find that our pattern recognition algorithm is able to find the high-frequency patterns for all benchmarks. It is also

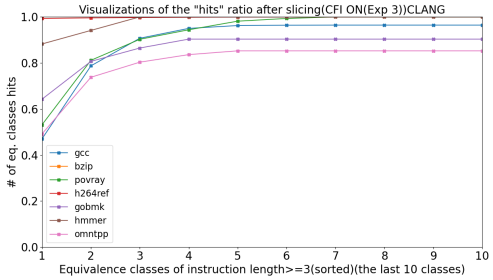
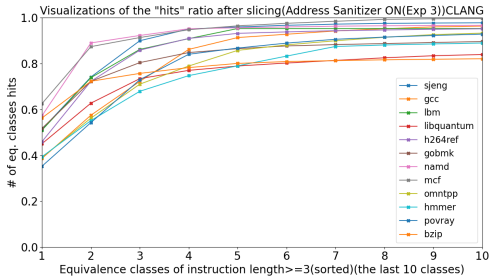
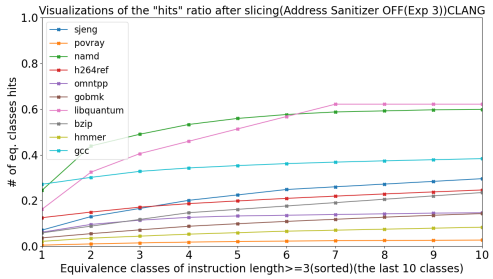


Figure 4: Pattern recognition results for SET-A-CFI



(a) AddressSanitizer ON (SET-A)



(b) AddressSanitizer OFF (SET-B)

Figure 5: Pattern recognition results for AddressSanitizer

encouraging to find identical high-frequency patterns being detected across all benchmarks. We do not plot the graph for the SET-B configuration with CFI since our techniques to prune spurious snippets eliminate most of the instances, as seen from Table 5.

5.2.3 Security Check Pattern for AddSan

Figure 5(a) illustrates the sorted cumulative *match ratios* over sliced and validated code snippets for the SET-A-AddressSanitizer (security check turned ON) benchmark configuration. Again we find that our technique is very effective at finding the common instruction patterns, corresponding to the address sanitizer security check, for the SET-A configuration. For the Set-B configuration with Address Sanitizer (Figure 5(b)), our algorithm does not find any high-frequency patterns for any benchmarks, which supports the likelihood that these are spurious instances

Table 8: Security Check patterns

Security Check	Compiler	Pattern Observed
STACKGUARD	CLANG	MOV RXX,qword ptr FS:[0x0] CMP RXX,qword ptr [RSP] J 0x0 CALL 0x0
CFI	CLANG	CMP RXX,RXX J 0x0 UD2
Address Sanitizer	CLANG	SHR RXX,0x0 MOV RXX,byte ptr [RXX + 0x0] TEST RXX,RXX J 0x0 AND RXX,0x0 ADD RXX,0x0 J 0x0

detected due to conservative filtering algorithms for the security check turned OFF case.

5.2.4 Common Patterns for Set-A Configs

Table 8 lists the most common instruction patterns seen across all the benchmarks in the SET-A configurations with the Stackguard, CFI and Addresssanitizer security checks. We found a high-frequency instruction pattern (or equivalence class) consistently present across all benchmarks in the SET-A configurations. It is both encouraging and notable that, on manual inspection of the binaries in SET-A, we find that these instruction patterns actually correspond to the security check instructions inserted by the compiler for each respective check. Furthermore, with the SET-B configuration when the check was disabled, no consistent dominating instruction pattern was detected across all benchmarks.

It is important to realize that filtering the program instructions from the validated snippets that is achieved by our slicing-based extension is significantly consequential to such precision in the pattern matching results. Thus, for the automated run-time security checks investigated in this study, our slicing based extension enables this algorithm to effectively discern when a security check is enabled or disabled in a given program binary.

6 Conclusion

Our primary objective in this work is to propose and evaluate the benefit of a novel program-slicing based extension to an earlier approach devised to detect the presence of run-time security checks in arbitrary software binaries. Our slicing based extension identifies and separates the binary-level instructions into *program* and *non-program* categories. We employ and

integrate our algorithm to remove program instructions from potential security-check snippets extracted by this earlier approach.

We explain and evaluate our technique for SPEC benchmarks compiled with two compilers for three different security checks. We found that our slicing-based approach is highly consequential in improving the ability and accuracy of this earlier technique to determine the presence of security checks in program binaries. We anticipate that our work will greatly enhance automated and independent security analysis of binary code, particularly for end-users who do not have access to the source code.

REFERENCES

Abijah Roseline, S. and Geetha, S. (2021). A comprehensive survey of tools and techniques mitigating computer and mobile malware attacks. *Computers Electrical Engineering*, 92:107143.

Adhikari, A. and Kulkarni, P. A. (2022). Using the strings metadata to detect the source language of the binary. In Daimi, K. and Al Sadoon, A., editors, *Proceedings of the ICR'22 International Conference on Innovations in Computing Research*, pages 190–200, Cham. Springer International Publishing.

Brooks, T. N. (2018). Survey of automated vulnerability detection and exploit generation techniques in cyber reasoning systems. In *Science and Information Conference*, pages 1083–1102. Springer.

Cadar, C., Dunbar, D., Engler, D. R., et al. (2008). Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224.

Cha, S. K., Avgerinos, T., Rebert, A., and Brumley, D. (2012). Unleashing mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, page 380–394, USA. IEEE Computer Society.

CVE (2019). A buffer overflow vulnerability in whatsapp voip stack.

Cybersecurity, U. and Agency, I. S. (2021). Top routinely exploited vulnerabilities.

Database, N. N. V. (2021). Cvss severity distribution over time.

De Lucia, A. (2001). Program slicing: methods and applications. In *Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149.

Eschweiler, S., Yakdan, K., and Gerhards-Padilla, E. (2016). discovre: Efficient cross-architecture identification of bugs in binary code. In *NDSS*, volume 52, pages 58–79.

Galindo, C., Pérez, S., and Silva, J. (2022). Program slicing techniques with support for unconditional jumps. In Riesco, A. and Zhang, M., editors, *Formal Methods and Software Engineering*, pages 123–139, Cham. Springer International Publishing.

Gao, D., Reiter, M. K., and Song, D. (2008). Binhunt: Automatically finding semantic differences in binary programs. In *International Conference on Information and Communications Security*, pages 238–255.

Henning, J. L. (2006). Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17.

Howard, M. and Lipner, S. (2006). *The Security Development Lifecycle*. Microsoft Press, USA.

Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., and Song, D. (2014). Code-Pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 147–163, Broomfield, CO. USENIX Association.

National Security Agency ghidra, N. (2019). Ghidra. https://www.nsa.gov/resources/everyone/ghidra/.

NIST (2022). National Vulnerability Database. https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cvss-severity-distribution-over-time.

OpenSSF (2024). Open source security foundation (openssf) best practices working group: Compiler options hardening guide for c and c++.

Pramanick, K. and Kulkarni, P. A. (2022). Detect compiler inserted run-time security checks in binary software. In Su, C., Gritzalis, D., and Piuri, V., editors, *Information Security Practice and Experience*, pages 268–286, Cham. Springer International Publishing.

Qasem, A., Shirani, P., Debbabi, M., Wang, L., Lebel, B., and Agba, B. L. (2021). Automatic vulnerability detection in embedded devices and firmware: Survey and layered taxonomies. *ACM Comput. Surv.*, 54(2).

Sarda, S. and Pandey, M. (2015). *LLVM Essentials*. Packt Publishing.

Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. (2012). Addresssanitizer: A fast address sanity checker. In *USENIX ATC 2012*.

Szekeres, L., Payer, M., Wei, T., and Song, D. (2013). Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, page 48–62.

Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., and Pike, G. (2014). Enforcing Forward-Edge Control-Flow integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955, San Diego, CA. USENIX Association.

Ucci, D., Aniello, L., and Baldoni, R. (2019). Survey of machine learning techniques for malware analysis. *Computers Security*, 81:123–147.

Weiser, M. (1981). Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, page 439–449. IEEE Press.

Wheeler, D. A. (2014). Preventing heartbleed. *IEEE Computer*, 47(8):80–83.

Wired (2017). The reaper iot botnet has already infected a million networks.

Zhang, Y. (2019). Sympas: Symbolic program slicing.