Explore Capabilities and Effectiveness of Reverse Engineering Tools to Provide Memory Safety for Binary Programs*

Ruturaj Vaidya[1], Prasad A. Kulkarni[1], and Michael R. Jantz[2]

[1] University of Kansas, USA ruturajkvaidya@ku.edu; prasadk@ku.edu
[2] University of Tennessee, USA mjantz@utk.edu

Abstract. Any technique to ensure memory safety requires knowledge of (a) precise array bounds and (b) the data types accessed by memory load/store and pointer move instructions (called, *owners*) in the program. While this information can be effectively derived by compiler-level approaches much of this information may be lost during the compilation process and become unavailable to binary-level tools. In this work we conduct the first detailed study on how accurately can this information be extracted or reconstructed by current state-of-the-art static reverse engineering (RE) platforms for binaries compiled with and without debug symbol information. Furthermore, it is also unclear how the imprecision in array bounds and instruction owner information that is obtained by the RE tools impacts the ability of techniques to detect illegal memory accesses at run-time. We study this issue by designing, building, and deploying a novel binary-level technique to assess the properties and effectiveness of the information provided by the static RE algorithms in the first stage to guide the run-time instrumentation to detect illegal memory accesses in the decoupled second stage. Our work explores the limitations and challenges for static binary analysis tools to develop accurate binary-level techniques to detect memory errors.

1 Introduction

Buffer overflow attacks rely on exploiting illegal memory accesses by referencing a buffer outside its legal array bounds. These attacks are mostly caused by bugs in software written in low-level memory unsafe languages, like C or C++ [37]. Such memory errors present an old security issue that persists in spite of advanced exploit mitigation mechanisms and can lead to silent data corruption, security vulnerabilities, and program crashes. In spite of solutions proposed through techniques at the programmer/source-level [16,24], compiler-level [2,4,7,9,14,27,29], and binary-level [33,36,38], the problem of memory safety persists especially in embedded, low-level, performance critical, and legacy software systems.

Techniques to detect memory errors require the ability to determine accurate buffer bounds along with the data type referenced (called the *owner* in this

* We thank the anonymous reviewers and the paper shepherd. This work is sponsored in part by the National Security Agency (NSA) Science of Security Initiative.

work) by each memory access (read/write) and pointer assignment/move instruction. This information is largely available to the source-code and compiler-level techniques, and enables more precise memory error detection at run-time. Unfortunately, techniques at this level require access to the source code and may not be applicable to legacy software where source code may not be available. Such techniques also involve reprogramming and/or re-compiling the code. The *single* binary executable generated/deployed using these techniques cannot be easily adapted to different risk averseness and performance overhead tolerances of end-users. Such approaches also leaves the task of memory safety solely in the hands of the software developer (rather than the end-user).

Binary-level techniques can overcome these challenges of source-level approaches. However, much of the program syntax and semantic information needed by techniques to resolve memory errors may be lost during the compilation process, especially when the generated binary is *stripped* of debug symbols. To overcome this limitation for binary-level techniques, researchers have developed advanced reverse engineering (RE) frameworks with sophisticated disassemblers, decompilers, and binary type and symbol inference algorithms that attempt to reconstruct information lost during the source to binary translation process.

In this work we study how much of the array bounds and instruction owner information is preserved by the compilation process (for binaries generated with *debug* information and those *stripped* of debug symbols) and can be retrieved by traditional *disassemblers* provided with contemporary RE tools. We also conduct the first detailed study on how accurately can this information that is needed to detect/prevent memory errors be reconstructed by the advanced *decompilers* and type inference algorithms provided with modern RE frameworks for *stripped* binaries. Our work explores the capabilities of two state-of-the-art RE tools, specifically NSA's Ghidra [28] and Hex-Ray's IDA Pro [1], and assesses the accuracy of the information they derive from program binaries.

Imprecision in array bounds detection and instruction owner information obtained by static RE tools can affect the ability to detect and prevent buffer overflows at run-time. In this work, we design and build a new binary-level run-time tool to evaluate, for the first time, the effectiveness of the program information gathered by the RE frameworks (in different configurations) to detect and prevent memory errors. The tool uses the obtained static analysis information to keep track of owners as pointers are assigned, and check relevant buffer reads/writes to assess the ability to ensure fine-grained memory safety at run-time.

Thus, we claim two major contributions in this work.

1. We conduct the first detailed study to determine the ability of static RE tools, specifically Ghidra and IDA Pro, to derive precise array bounds and instruction *owner* information from binary programs, which is required to detect and prevent memory errors.
2. We design, build and employ a new decoupled binary-level execution-time tool with the goal to assess the efficacy of the statically derived program information to provide memory safety for binary programs.

2 Related Works

In this section we compare our work with studies that evaluate the capabilities and precision of reverse engineering frameworks to reconstruct program information lost during the translation process. We also discuss past research in binary-level techniques to detect and prevent memory errors.

Several prior research works have evaluated the accuracy of binary code disassemblers and decompilers. Meng and Miller identify challenging code constructs that make it hard for RE tools to accurately disassemble binary code and construct a correct control flow graph (CFG) [25]. Andriesse et al. compare 9 popular disassemblers and find that complex code constructs are rare in real-world programs [3]. Inaccuracy in function start/boundary detection by current RE tools was reported by some works [3, 5]. Pang et al. analyze 9 open-source disassemblers to compare the algorithms and heuristics used for instruction recovery, symbolization, function detection and CFG construction and assess their precision [31]. They find that different tools use distinct algorithms and heuristics that complement each other, but also introduce coverage-correctness trade-offs. Another study explores the usability and effectiveness of decompilers to recover C output from binary code [21]. They find that while modern decompilers are getting increasingly powerful and accurate, issues such as type recovery and optimization still impede decompilers from generating accurate and presentable outputs. None of these works assess the efficacy of array bounds and instruction owner detection in RE tools.

A plethora of research has been conducted on type inference from program binaries [6, 13, 17, 19, 20, 23, 30, 35, 39]. Most of these research efforts are focused on prediction of basic or preliminary type information. Although some of these approaches claim to be able to detect higher order structures or aggregate types likes arrays, none of the approaches we know assess the accuracy of array bound detection, or evaluate the precision of instruction owner detection for binaries.

Past researchers have developed many techniques to detect and prevent memory errors. Many past approaches rely on the source-code with access to rich semantic program information [2, 8, 10, 27, 29, 32, 34].

Binary-level tools to locate fine-grained buffer overflows in memory at runtime have also been developed. The BinArmor technique [36] to detect memory errors relies on a tool called *Howard* [35] that uses past program execution traces to extract data structures and their memory bounds. BinArmor uses information from Howard to statically instrument the binary with checks to detect unsafe memory accesses during later program executions. Another technique develops a memory layout recovery algorithm to locate memory access vulnerabilities in the program *after* execution of the failed run [38]. This approach requires traces from a set of correct program executions to recover fine-grained memory layouts of variables. The recovered memory layouts from the passed program executions are then used to determine if the failed run exceeded any valid variable boundaries.

Both these past techniques employ a dynamic approach that relies on traces from multiple correct prior program executions to determine or predict relevant properties about the program, including buffer bounds. All dynamic analysis

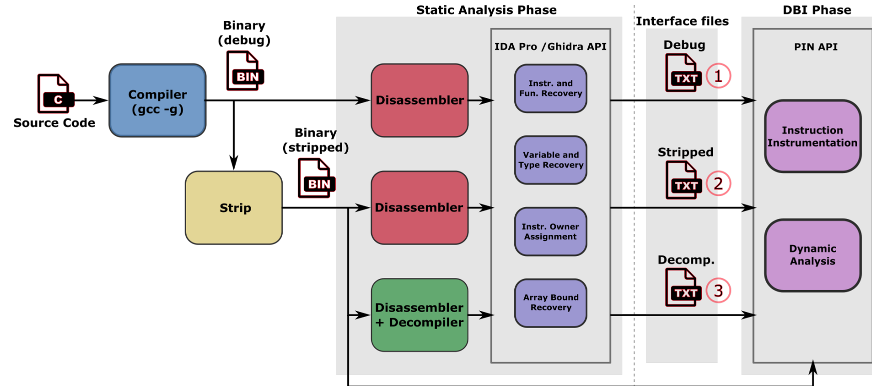


Fig. 1: Schematic of Experimental Framework Setup

techniques require representative program inputs and are incomplete by design since they cannot guaranty complete code coverage and can only protect code and buffers that were seen by the analyzed program execution traces. Instead, our work is the first to explore the potential, capabilities and trade offs of using a static analysis and static type inference based approach to resolve this problem. Similar to BinArmor, but unlike the approach by Wang et al., our technique is designed to detect memory errors before they are triggered during program execution. Most importantly, none of these tools are available for use by researchers in the open-source domain and none have attempted to employ these tools to assess the *extent* and *impact* of inaccuracies in array bounds and instruction owner detection to locate and prevent buffer overflows at run-time.

3 Benchmarks and Frameworks

In this section we describe the experimental setup, benchmarks, and tools and frameworks used for this study.

3.1 Experimental Framework

A schematic of the overall framework is illustrated in Figure 1. A C/C++ program is compiled with the standard `gcc` compiler with the “-g” flag to generate a binary with embedded debug symbol information. This binary is used by our ① **Debug** configuration. Later, the `strip --strip-all` Linux command is used to generate another binary executable that is *stripped* of all symbol information. This binary is used by our ② **Stripped** and ③ **Decomp.** configurations.

Our experiments employ two stages: (a) *static analysis* to assess the ability of our RE tools to derive precise array bounds and instruction owner information from binary programs, followed by (b) *dynamic binary instrumentation* (DBI) to assess the efficacy of the statically derived program information to provide memory safety for binary programs. We employ Ghidra version 9.1.2 (with Ghidra

decompiler) [28] and IDA Pro version 7.5 (with Hex-Rays decompiler) [1] to conduct static analysis in the first stage. We use the PIN (version pin-3.15) [22] dynamic instrumentation engine in the second stage. All experiments are performed on a cluster of x86-64 Intel Xeon processors with the Fedora 28 OS.

The static RE tools we employ work independently of the program input(s). They include a *disassembler* to convert machine code to assembly code. They also provide a *decompiler* that employs sophisticated type inference and code reconstruction algorithms to raise the low-level assembly code into a higher-level language representation (commonly, C). Our ① Debug and ② Stripped configurations only use the *disassembler*. The ③ Decomp. configuration also uses the *decompiler*, which enables this configuration to recover higher-order program structures like arrays and pointers along with their associated sizes and assists with instruction owner detection from the *stripped* binary. Each configuration in the static analysis phase outputs a distinct *interface file* with the array bounds and instruction owner information that it can recover from the binary.

The stripped binary program and the statically generated interface file are provided to Pin. Pin adds instrumentation based on previously determined instruction owner mapping and array bounds information, tracks dynamically allocated buffers and relevant register and memory values, and inserts security checks to detect buffer overflows at run-time.

3.2 Benchmarks

In this work we use benchmarks from three different benchmark suites, SARD-89 [18, 26], SARD-88 [26, 40], and SPEC cpu2006 [15]. The SARD-89 suite contains 291 small programs that implement a taxonomy of diverse C buffer overflows (1164 total programs). Each test case has three versions with memory accesses that overflow *just outside*, *moderately outside*, and *far outside* the buffer, respectively. The fourth version for each test case is a patched version without any buffer overflow. 18 of the 291 test subjects in SARD-89 benchmark suite contain overflows that leverage library functions to succeed. Although not a fundamental limitation of our technique or tools, we currently do not analyze library functions, and so leave out these programs. Additionally, 152 test subjects in SARD-89 overflow the buffer with an index that is a *constant* integer, for example `buf[2048]`. We discuss these cases in more detail in Section 4.2. We use the remaining 121 test programs for all experiments in this work, unless mentioned otherwise.

The SARD-88 benchmark suite contains 14 "real-world" programs from various internet applications (BIND, Sendmail, WU-FTP) with known buffer overflows. Two versions are provided for each test case, one with and the other without a buffer overflow (28 programs in total). We statically link library functions like `strcpy`, `strcmp`, that can overflow in some of the SARD-88 programs. We also employ all the SPEC cpu2006 integer benchmarks to study the scalability and efficacy of the static tools on large programs. All benchmarks are compiled using GCC version 9.3.1; optimized benchmarks use `-O3`.

4 Static Reverse Engineering

Techniques to detect and prevent memory errors need precise information regarding buffer data types, their base address and size/bound, and the data type referenced (*owner*) by each memory access (read/write) and pointer assignment/move instruction. Much of this information is lost during the compilation process. RE frameworks employ complex algorithms and heuristics to reconstruct lost program information from binaries. We explored the abilities of several RE tools to identify and reconstruct program information that is required to detect and prevent memory-related attacks in binaries, including Angr, Radare/r2, Debin, Ghidra, and IDA Pro. We found that only Ghidra and IDA Pro provide the capability and API for this task. In this section we present our results and observations. To our knowledge, this is the first work that evaluates and reports the efficacy of RE tools to extract or reconstruct the buffer/pointer bound and instruction owner information required to detect/prevent memory errors.

4.1 Setup and Implementation Details

In this section we describe the algorithms and extensions we develop to explore the capabilities of Ghidra and IDA Pro. Our scripts extract information relating to the statically known object bounds (local/global variables) and *instruction-owner* mappings. We use the term *owner* for program variables of type array or pointer that constitute the memory operand for the memory access instructions (of the kind `MOV` for the x86-64). Additionally, we have also extended the tools with block-level data-flow algorithms to track the instructions that propagate the *pointer* variables from memory to registers before they are used.

Figure 2 illustrates the information we gather from our RE tools. The figure shows the source code, the compiler generated binary code and corresponding IDA Pro output for a simple C program. This program has a single integer buffer, '`b`', an integer pointer, '`ptr`', and an integer scalar '`n`'. The variable '`ptr`' is the "*owner*" of the assembly instructions at offsets '`8`', '`20`' and '`27`'. '`ptr`' is mapped to the corresponding addresses. The pointer access on line #6 overflows the array '`b`' - corresponding to assembly instruction at offset '`27`'. Comparably, the direct array access on line #7 overflows the array '`b`' - corresponding to assembly instruction at offset '`32`'. Memory safety algorithms need to check such accesses to determine the invalid access at run-time.

We found that the *owners* of direct variable access instructions (that employ $\{rbp, rsp, rip\}$ based relative addressing, like the instructions at address '`8`', '`20`' and '`32`' in Figure 2) are determined automatically by the reverse engineering frameworks we study. However, the *owners* of pointer dereference instructions (for example, the instruction at address '`27`' in Figure 2) are not detected automatically by our advanced tools. To analyze such memory accesses, we implement a simple data-flow algorithm that keeps track of the variables and owners as they move between the memory stack and registers.

Figure 2(c) shows the output of our RE scripts after analysing the binary generated using the example program shown in Figure 2(a). This output file

```c
1  int foo()
2  {
3    int b[5], n;
4    int *ptr = b;
5    n = 10;
6    *(ptr+n) = 4;
7    b[n] = 9;
8    return 0;
9  }
```

```
1  0: push rbp
2  1: mov rbp,rsp
3  4: lea rax,[rbp-0x20]
4  8: mov QWORD PTR [rbp-0x8],rax
5  c: mov DWORD PTR [rbp-0xc],0xa
6  13: mov eax,DWORD PTR [rbp-0xc]
7  16: cdqe
8  18: lea rdx,[rax*4+0x0]
9  1f:
10 20: mov rax,QWORD PTR [rbp-0x8]
11 24: add rax,rdx
12 27: mov DWORD PTR [rax],0x4
13 2d: mov eax,DWORD PTR [rbp-0xc]
14 30: cdqe
15 32: mov DWORD PTR [rbp+rax*4-0x20],0x9
16 39:
17 3a: mov eax,0x0
18 3f: pop rbp
19 40: ret
```

```
1  1          # num. functions
2  foo        # fn name
3  0          # fn start
4  40         # fn end
5
6  32         # stack size
7  addresses  # owner-ins mapping
8  8 foo_ptr
9  20 foo_ptr
10 27 foo_ptr
11 32 foo_b
12
13 locals     # locals info.
14 -32 ARRAY foo_b 20
15 -12 scalar foo_n 4
16 -8 PTR foo_ptr 8
17
18 .global    # globals info.
```

Fig. 2: Example showing an invalid array access: (a) C source code (b) Assembly output (c) Output text file (interface file) after static analysis by IDA Pro

contains function related metadata such as *owner*-instruction address mapping I (listed under *addresses*), function variable metadata f_v - local variables along with their position (offset) on the stack relative to the stack pointer, their size and type (listed under *locals*), function boundary $(f_s \cup f_e)$, and additional metadata f_m such as number of functions, stack size, base pointer relative addressing information, etc. This file also contains global variable metadata G_v - Variables defined in the *data* or *bss* sections and associated with their static address; the rest of the metadata is similar to local variables (listed under *.global*). This output of the static analysis $G_v \cup \sum f_i\{(f_s \cup f_e), f_v, f_m, I\}$ is fed to the Pin tool.

4.2 Efficacy of Reverse Engineering Tools

In this section we study the efficacy of existing reverse engineering tools to determine buffer bound and instruction owner information for programs compiled by standard compilers with and without debug symbols and compiler optimizations.

Failures Even with Debug Symbol Information. Building a binary with debug symbols retains useful information from the source program regarding the function *stack* and the global *data/bss* section layout, variable types, and buffer bounds. However, the owner information is not captured by the debug symbols and may become hard to infer from the static binary. An example of this challenging scenario is encountered for many SARD-89 benchmarks that overflow the buffer *with an index that is a constant integer*. An example of this case is illustrated in Figure 3. The left-hand side of the figure shows the source code and the right-hand side shows the corresponding assembly code. This program declares two arrays, '`b1[5]`' and '`b2[10]`'. The write to '`b2[15]`' corresponds to assembly instruction at location '`40112e`' and the read from '`b1[3]`' corresponds to the assembly instruction at location '`401135`'. In the assembly code these buffer accesses that are *indexed by a constant* use a displacement relative to the stack frame pointer, `rbp`, rather than the base array pointer. Thus, although

```c
1 #include <stdio.h>
2 int main()
3 {
4   int b1[5];
5   int b2[10];
6   b2[15] = 1;
7   printf("%i\n", b1[3]);
8   return 0;
9 }
```

```
1  0000000000401126 <main>:
2  401126: push rbp
3  401127: mov rbp,rsp
4  40112a: sub rsp,0x50
5  40112e: mov DWORD PTR [rbp-0x14],0x1
6  401135: mov eax,DWORD PTR [rbp-0x14]
7  401138: mov esi,eax
8  40113a: mov edi,0x402010
9  40113f: mov eax,0x0
10 401144: call 401030 <printf@plt>
11 401149: mov eax,0x0
12 40114e: leave
13 40114f: ret
```

Fig. 3: Ambiguous array access: (a) C source code (b) Assembly output

these two instructions reference different buffers (and one, `b2[15]` is an overflow), if these accesses are within the stack frame, then it is hard for the RE tools to infer or predict from the assembly code if they refer to the array '`b1`' or '`b2`' or neither. In such cases, we found that the reverse engineering tools cannot determine the correct instruction *owner* even in the presence of debug symbols.

Such failures caused due to buffer accesses by a constant numeral may be an *intrinsic* limitation of binary-level techniques. Fortunately, arrays dereferenced by a constant numeral may be a less critical hazard or attack vector in security threat models, as many real-world buffer-overflow and stack-smashing attacks are triggered by a malicious external input specifically devised to overflow the buffer bound. Lack of high-level program information also prevents our RE tools from associating the correct *owner* with instructions accessing individual members of a structure. We found that there are 152 test cases in the SARD-89 benchmark suite that our RE tools fail to analyze due to these *intrinsic* reasons. We leave out these programs from the remaining experiments in this paper.

Accuracy of Type and Owner Detection for Arrays and Pointers Figures 4 and 6 (in Appendix A for optimized benchmarks) display the efficacy of array and pointer type detection for programs in the SARD-89, SARD-88, and SPEC suites. Each figure shows three configurations for each of our static RE tools, ① **Debug**, ② **Stripped** and ③ **Decomp**. We leverage the *pyelftools* [12] module to design and build a new tool to extract variable information directly from the "dwarf" [11] section of binaries [3]. The data from this tool is used as a baseline to compare the results obtained in the other RE-based configurations.

Figures in the first row (4(a), 4(e), 4(i)) display array bound detection accuracy for corresponding benchmarks. *#TP Arrays* show the (*True Positive*, TP) arrays detected at correct offsets regardless of their size/bound, while *#FP Arrays* show the (*False Positive*, FP) arrays that are detected at incorrect offsets compared to our baseline. Figures in the second row (4(b), 4(f), 4(j)) display the accuracy of pointer detection. The first set of bars in each of these figures show the number of TP and FP pointers as detected *directly* by the reverse engineering tools. The set of bars labeled "*with Pred.*" use a simple pointer prediction

[3] DWARF is a debugging file format used by many compilers, including the GCC compiler used in this work, to support source level debugging.

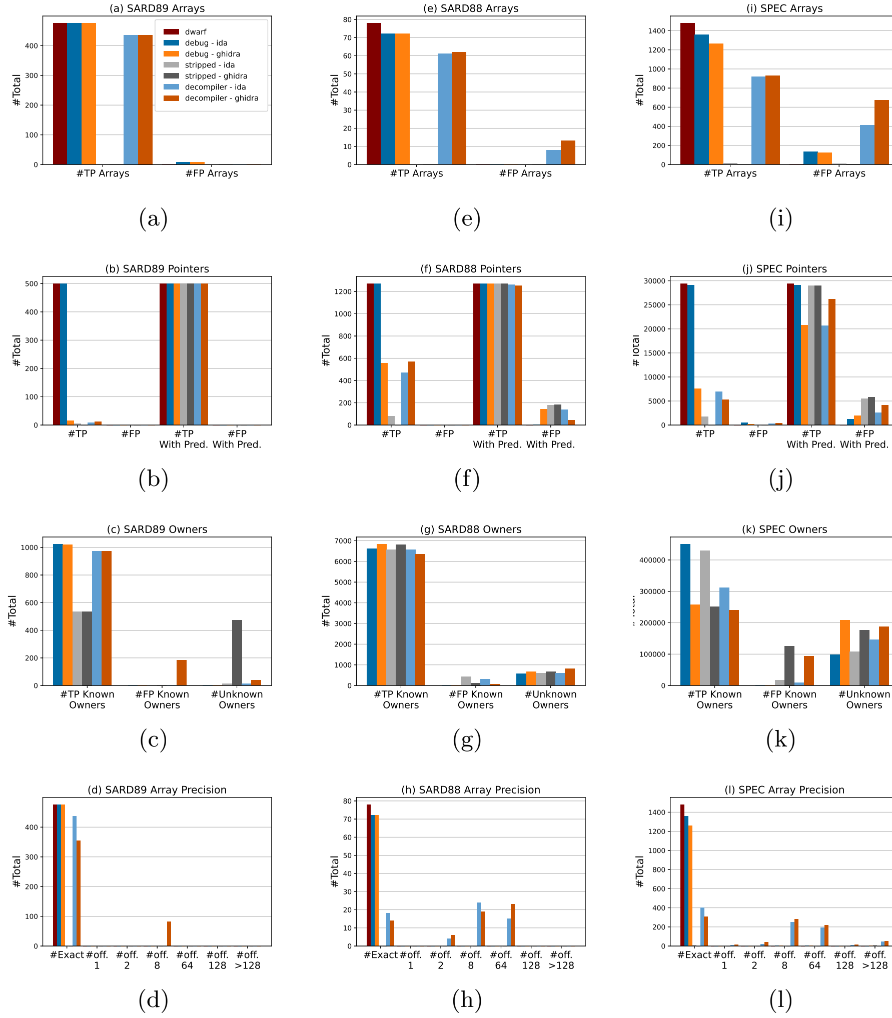


Fig. 4: Accuracy of array, pointers, and owner detection for SARD-89, SARD-88, SPEC-cpu2006 benchmarks, generated without compiler optimizations

algorithm we employed that marks every variable with “undefined type” (or undetermined type) and with a size of 8 bytes as a pointer. Figures in the third row (4(c), 4(g), 4(k)) display the accuracy of mapping the move and memory dereference instructions to array/pointer owners. The *known owners* are associated with instructions by our analysis algorithm. Static instructions mapped to owners that are scalar variables are ignored. Instructions are assigned *unknown owners* when relevant owners cannot be predicted. The *stripped* and *decompiler* results in these figures are compared against those when debug symbols are available with each respective RE tool. Finally, figures in the last row (4(d),

4(h), 4(l)) plot the accuracy of array bounds detection for the *#TP Arrays*. The Y-axis in these figures indicates the error magnitude in array bound detection.

Both Ghidra and IDA Pro show very poor efficacy with *optimized* programs for most of our experiments. One reason is that these RE tools do not consider register allocated variables which are prevalent in optimized benchmarks. This observation suggests a critical area for future work. Given this state with optimized binaries, we focus on unoptimized benchmarks for the remainder of this section. Results for optimized binaries are presented in Figure 6 in Appendix A.

We make the following interesting observations from the data presented in Figure 4: (1) Even advanced RE tools, like Ghidra and IDA Pro, can fail to appropriately leverage program symbol information, as seen most prominently by the poor efficacy of the *debug-ghidra* configuration to accurately detect static program pointers. (2) The *debug-ghidra* configuration misrepresents pointer types as `int_64` or `undefined_64` in many cases. (3) With no symbol information available in stripped binaries, *disassemblers* in our RE tools are unable to detect most/all arrays and pointers. Surprisingly, our simple pointer prediction algorithm is able to correctly detect most *true* pointers but also produces many false positives. We will explore developing more sophisticated pointer detection algorithms in the future to improve this simple prediction model. (4) *Decompiler* algorithms in IDA Pro and Ghidra do a commendable job, especially in detecting arrays and array bounds in stripped binaries. Interesting, even small programs seem to be able to provide sufficient context information to enable effective array type detection with these algorithms. (5) We find that the decompilers in Ghidra and IDA Pro are more accurate in inferring arrays and array bounds than inferring pointers. However, decompiler-based type inference algorithms often split arrays or combine them with adjacent arrays/variables resulting in many false positives and at times large inaccuracies in bound detection. (6) We can also make some more specific observations, like in Figure 4(c) for SARD-89 benchmarks, many instructions associated with a scalar in the stripped-IDA case (which are ignored and are not plotted in the figure) are not assigned any owner (*unknown* owner) in the stripped-Ghidra case. We will see the implication of this difference in the next section. While instruction owner detection appears to works well for unoptimized benchmarks, it is largely unsuccessful for optimized programs. These observations reveal both the current capabilities of the static RE tools, Ghidra and IDA Pro, and open areas for research to more accurately derive program information that is necessary to detect memory errors at run-time.

5 Run-time Framework to Detect Memory Errors

In this section we describe the implementation details of our run-time framework that employs the static program information gathered from the RE tools to detect spatial memory errors. We also assess the efficacy of the complete framework to effectively use the program information extracted by the static RE tools to detect memory errors in programs during execution. This approach does not require access to source code or hardware support.

Algorithm 1: Run-time Object overflow detection

```
Input: Function Metadata F → ∑ f_i{(f_s ∪ f_e), f_v, f_m, I}
Input: Global Metadata G → G_v
1  ReadInput(Input);
2  InstrumentMallocFree();
3  F_d ⟶ Set of functions reached during execution;
4  I_d ⟶ Instructions mapping per function reached during execution;
5  foreach f ∈ F_d do
6      foreach i ∈ I_d do
7          if i.Address == f_s then InitializeStack() ;
8          if i.Address == f_e then UnInitializeStack() ;
9          if i.Owner ∈ Unknown then UnknownBoundCheck(); continue ;
10         if IsInsMemStore() then
11             if i.Owner ∈ Pointer then
12                 if i.BaseReg ⊆ {rbp, rsp, rip} then BoundPropogationCheck();
13                 else PtrBoundCheck();
14             end
15             else ObjBoundCheck();
16         end
17         else if IsInsMemLoad() then
18             if i.Owner ∈ Pointer then PtrBoundCheck();
19             else ObjBoundCheck();
20         end
21     end
22 end
```

5.1 Dynamic Tracking and Instrumentation using Pin

Pin [22] employs information supplied by the our static RE tools in the *interface file* to detect memory safety violations at run-time. We build scripts, called *Pintools*, that use the Pin API to insert dynamic checks in the executed code. Algorithm 1 explains our dynamic buffer overflow detection algorithm. Our Pintool will add instrumentation code at run-time for pointer/array memory move/dereference instructions that are mapped with corresponding instruction owners from the interface file. Run-time instrumentation is added for the *static* instruction categories mentioned below. We employ the example program in Figure 2 to explain the run-time algorithm and illustrate the instrumentation categories.[4]

I. Function start: The `InitializeStack()` function in Algorithm 1 will add instrumentation code at each function prologue to mark the locations of local variables w.r.t. the actual value of the stack pointer in memory. Function start (f_s) address obtained from the interface file (Figure 2(c) - line #3) determines the instrumentation point. The dynamic array/pointer variable locations and available bounds get stored in a global metadata structure in this phase. Arguments passed to the program are also detected in this phase by adding a special check for function '`main`'.

II. Function end/return: The `UnInitializeStack()` function in Algorithm 1 will fetch the function end (f_e) address from the interface file (Figure 2(c) - line #4) to add instrumentation at every function end. This type of instrumentation is required to roll-back the allocated stack and remove corresponding meta-data when the function returns.

[4] Our code can be accessed here: https://github.com/Ruturaj4/vulcan_prototype

III. Pointer move/propagate: This type of instrumentation is used to transfer and assign the address/bound of the buffer to any associated pointer. The pointer can then be used to indirectly access the buffer. Similarly, bounds can also be transferred between two pointers. Instructions at offsets '4' - '8' (from Figure 2(b)) give an example instruction pattern that represents pointer propagation.

```
lea rax,[rbp-0x20]
mov QWORD PTR [rbp-0x8],rax
```

Here, the `lea` (load effective address) instruction computes the address of buffer '`b`' into a register (`rax`), and then assigns it to the pointer '`ptr`' (at offset '(`rbp-0x8`)' on the stack). Thus, the static analysis tools mark the owner of instruction at offset '8' as pointer '`ptr`' (line #8 in Figure 2(c)).
At run-time, the `BoundPropogationCheck()` function in our Pintool will add instrumentation code (to the store instruction at offset '8' in Figure 2(b)) to check the contents of the `rax` register to determine the location of object '`b`' in memory. Note that the address and bounds of '`b`' get stored in a global map structure during stack initialization at function start. It will then transfer these bounds to the pointer '`ptr`'.
IV. Pointer dereference: The following instruction triplet (instructions at offset '20' - '27' from Figure 2(b)) shows an example pattern for pointer dereference.

```
mov rax,QWORD PTR [rbp-0x8]
add rax,rdx
mov DWORD PTR [rax],0x4
```

The buffer '`b`' is accessed through pointer '`ptr`'. Here, the `PtrBoundCheck()` function from Algorithm 1 will add instrumentation code (just before the store instruction at offset '27') to check whether the access is within the associated bounds, as follows:

```
if (access < low_bound || access >= up_bound)
    abort;
```

V. Array/Object bound check: Similar to `PtrBoundCheck()`, the `ObjBoundCheck()` function adds code to verify that a direct array access is within the associated bounds. An example pattern (instruction at offset '32' in Figure 2(b)) is:

```
mov DWORD PTR [rbp+rax*4-0x20],0x9
```

VI. Memory accesses with unknown instruction owner: In some cases our static RE tools are unable to determine the instruction owners for the memory access instructions in the binary. In such cases, the `UnknownBoundCheck()` function will add instruction code to check that the access is within the bounds of the current function stack.

Apart from the above instrumentation categories, we instrument dynamic memory allocation functions like `malloc`, `calloc`, etc. We use Pin's *routine instrumentation support* to instrument these dynamic allocation functions. Our implementation also supports pointer metadata propagation through function calls, i.e. it propagates the pointer bounds information whenever pointers are passed between different functions.

5.2 Buffer Overflow Detection Accuracy

The efficacy of this framework to accurately detect memory errors is influenced by two factors: (a) the ability of the employed static RE tools in the first stage to correctly discover the necessary program information, and (b) the ability of the dynamic Pin-based run-time framework to correctly detect the program patterns that constitute valid instrumentation points. The run-time framework should also maintain and correctly propagate the desired program state at the relevant instrumentation points.[5]

We check the effectiveness of our prototype framework to detect memory overflows using two benchmark suites – SARD-89 and SARD-88. Table 1 presents the efficacy of the framework with the SARD-89 benchmarks. Each SARD-89 benchmark consists of four programs, one that is categorized as *benign* (no overflow), and three categorized as *Malicious* with a memory reference that overflows some buffer with a *Minimum*, *Medium*, or *Large* amount, respectively.

Tables 1(a) and 1(b) show the efficacy results for the 121 SARD benchmarks that overflow for an instruction with a *non-constant array access*, with static analysis conducted by IDA Pro and Ghidra, respectively. For each configuration and benchmark, the column labeled *Basic* lists the number of programs that behave correctly or as expected (no-overflow or overflow detected at correct location) with our mechanism that does not add any instrumentation for instructions associated with *unknown* owners. The columns labeled *Ext.* display the results with the small extension to our run-time algorithm to add instrumentation for instructions with *unknown* owners to detect an overflow *if the access is outside the bounds of the current stack*.

Thus, we can see that, (a) All *Benign* cases are correctly handled. (b) All cases with the *Debug* configuration are correctly detected. (c) Most *Malicious* cases with the *Stripped* configuration cannot be detected due to missing information from the static analysis phase. The run-time Pin extension enables the detection of overflows outside the stack bounds for binaries analysed by Ghidra (that contain instructions with unknown owners). This extension does not help binaries analyzed by IDA Pro as it assigns *some* owner (a scalar in many cases) to all such relevant instructions. (d) Interestingly, advanced type and bounds detection conducted by the static tools enables the *Decomp.* configuration to correctly detect a large majority of overflows for the *Malicious* programs.

Table 2 presents the efficacy results for the 14 SARD-88 benchmark programs with the IDA Pro RE tool used in the first stage.[6] For each SARD-88 benchmark, the program with the odd number is *malicious* and contains a buffer overflow and

[5] The implementation of our run-time framework can correctly process all programs in the SARD-88 and SARD-89 suites, as well as most of the SPEC cpu2006 integer benchmarks. However, our implementation currently encounters memory/performance issues with some larger SPEC benchmarks. We will address these implementation issues and improve tool robustness in our ongoing work.

[6] The results with Ghidra in the first stage are similar, and are included in the Appendix in Table 3 to conserve space. There are more failures in the Ghidra-based configuration primarily due to poorer analysis of global strings and buffers by Ghidra.

Table 1: SARD-89 run-time results for three experimental configurations: ① Debug, ② Stripped ③ Decomp. (Stripped + decompiler)

	Benign			Malicious								
				Minimum			Medium			Large		
	#Total	Basic	Ext.	#Total	Basic	Ext.	#Total	Basic	Ext.	#Total	Basic	Ext.
Debug	121	121	121	121	121	121	121	121	121	121	121	121
Stripped	121	121	121	121	1	1	121	1	1	121	1	1
Decomp.	121	121	121	121	110	110	121	110	110	121	110	110

(a) Benchmarks with non-constant array accesses (IDA Pro)

	Benign			Malicious								
				Minimum			Medium			Large		
	#Total	Basic	Ext.	#Total	Basic	Ext.	#Total	Basic	Ext.	#Total	Basic	Ext.
Debug	121	121	121	121	121	121	121	121	121	121	121	121
Stripped	121	121	121	121	1	29	121	1	42	121	1	118
Decomp.	121	121	121	121	95	95	121	110	115	121	110	118

(b) Benchmarks with non-constant array accesses (Ghidra)

the program with the even number is *benign*. All results displayed here include the Pin extension to detect memory access beyond the current function stack.

We find that while most cases with the *Debug* configuration are detected correctly, there are a few notable failures. Most of these failures are due to incorrect static bound detection for *global* read/write buffers. We did not encounter this case in SARD-89 benchmarks; most overflows there were in *local* buffers.

Programs analyzed by Ghidra encounter additional failures, even in the *Debug* case, because, unlike IDA Pro, Ghidra does not detect global strings that are usually defined in the binary's read-only (`.rodata`) section. For instance, benchmarks II, IX, XI and XIV fail when analyzed by Ghidra due to this issue. We also observed that global read-only strings with lengths less than 4 bytes are not detected by IDA Pro; for Ghidra this length is 5 bytes. This issue is a basic limitation for reverse engineering tools, as reducing this lower bound can lead to type detection conflicts with other types that may appear to be strings.[7].

As expected, malicious programs in the *Stripped* configuration fail due to incorrect static analysis. However, in contrast to our observation that the *benign* cases with the *Stripped* configuration in SARD-89 are successful (no overflow detected), we find that most *benign-Stripped* cases in SARD-88 fail (false positive overflow is detected). This difference in behavior is because our RE tools make no owner association (or *unknown* owner with Ghidra) for the SARD-89 programs in this configuration; so, no check is added for programs analyzed by IDA Pro, and the only check added is to detect out-of-stack overflows for binaries analyzed by Ghidra. In contrast our RE tools associate an owner (global variables in many cases) with incorrect bounds (1 in many cases) for many SARD-88 programs in this configuration; hence, they encounter a false positive overflow.

[7] https://github.com/NationalSecurityAgency/ghidra/issues/2274

Table 2: SARD-88 test Results (IDA Pro) for our three experimental configurations: ① Debug, ② Stripped, and ③ Decomp. (Stripped + Decompiler)

Bechmarks		Debug	Stripped	Decomp.	Benchmarks		Debug	Stripped	Decomp.
I	283	✓	✗	✗	VIII	297	✗	✗	✗
	284	✓	✗	✓		298	✗	✗	✗
II	285	✓	✗	✓	IX	299	✓	✗	✓
	286	✓	✗	✗		300	✓	✗	✓
III	287	✓	✗	✗	X	301	✗	✗	✗
	288	✓	✗	✗		302	✓	✗	✓
IV	289	✓	✗	✗	XI	303	✓	✗	✗
	290	✓	✗	✗		304	✓	✗	✗
V	291	✓	✗	✓	XII	305	✗	✗	✗
	292	✓	✗	✓		306	✓	✓	✓
VI	293	✓	✗	✓	XIII	307	✗	✗	✗
	294	✓	✗	✓		308	✓	✓	✓
VII	295	✓	✗	✓	XIV	309	✓	✗	✓
	296	✓	✓	✓		310	✓	✗	✓

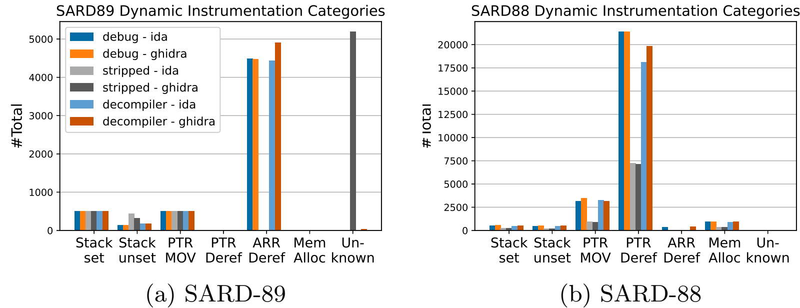
Again, we notice that advanced array bound and type inference enables several programs to be correctly handled in the *Decomp.* configuration. Of the 23 programs that are correctly detected in the *Debug* case, 15 are also correctly handled in the *Decomp.* configuration.

5.3 Performance Overhead

Figure 7 uses different metrics to estimate the performance overhead of the run-time framework.[8] Apart from the slowdown introduced by the Pin framework itself, the instrumentation added by our run-time algorithm is the primary source of performance overhead. Figures 5(a) and 5(b) plot the total number of instrumentation points encountered by all the SARD-89 and SARD-88 programs at run-time, respectively. The figures also highlight some interesting observations, including, (a) the number of *Stack sets* is less than the number of *Stack unsets* due to many programs exiting abruptly after an overflow is detected, (b) while SARD-89 programs are dominated by array dereferences, the SARD-88 programs encounter many more pointer dereferences, (c) the Ghidra-stripped configuration assigns an *unknown owner* to several instruction in SARD-89, which enables the detection of *large* buffer overflows that exceed the current stack bounds.

We also compare the execution time of the benchmark in three settings, (a) native run, (b) using a minimal pintool that does not add any instrumentation, and (c) the pintool implementing our run-time algorithm (plots are available in Figures 7(a) and 7(b) in Appendix C). Each program is run for 15 times and the

[8] In theory, the performance of our run-time framework should be comparable with a compiler-based approach, like SoftBound [27]. Our run-time implementation is currently in the prototype stage and was designed to primarily explore the properties and potential of the static RE tools to detect memory errors in program binaries. As such, we have not yet explored performance optimizations and associated trade offs with memory error detection accuracy for the run-time framework.



(a) SARD-89

(b) SARD-88

Fig. 5: Dynamic instrumentation points (subfigures (a) and (b))

average execution time is plotted. Most programs in the SARD-89 and SARD-88 suites run quickly, with an average execution time of 0.99$msec$ and 1.17$msec$ for the native run, respectively. The startup overhead of the minimal Pin framework increases the average run-time to 213.71$msec$ for SARD-89 and 417.99$msec$ for SARD-88 programs, respectively. Finally, our run-time framework increases the overhead to 227.85$msec$ for SARD-89 programs and 450.62$msec$ for the SARD-88 programs.

6 Conclusions and Future Work

Our goal in this work is to analyze and evaluate the ability of current state-of-art static reverse engineering tools, especially Ghidra and IDA, to accurately determine the required program information from binary programs to enable the effective detection of memory errors during program execution. We find that both Ghidra and IDA include advanced algorithms for array bound and instruction owner identification as part of their decompiler framework. However, more advanced techniques and algorithms are needed to further improve their capabilities and precision, especially for optimized binaries. We built a Pin-based run-time tool that can use the information from the static RE tools to detect buffer overflows during execution. We found that while our run-time tool can detect a large fraction of memory errors in our benchmarks, the accuracy of the tool is directly proportional to the limitations in the available static program information.

The practicality of this approach to detect memory errors is limited by the accuracy and completeness of the static tools and the efficiency of the run-time framework. In the future we will explore the potential of different approaches, including other dedicated type inference mechanisms, new static algorithms, and combining static and dynamic analysis, to improve array bound and owner detection, especially for optimized binaries. We will also study techniques to improve the efficiency of our prototype run-time framework, including using a static binary rewriting system. Finally, we will experiment with a larger benchmark set to more comprehensively study the properties of this approach.

References

1. Hex-rays decompiler. https://www.hex-rays.com/products/decompiler/ (2020)
2. Akritidis, P., Costa, M., Castro, M., Hand, S.: Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors. In: USENIX Security Symposium. pp. 51–66 (2009)
3. Andriesse, D., Chen, X., van der Veen, V., Slowinska, A., Bos, H.: An in-depth analysis of disassembly on full-scale x86/x64 binaries. In: 25th USENIX Security Symposium (USENIX Security 16). pp. 583–600. Austin, TX (Aug 2016)
4. Austin, T.M., Breach, S.E., Sohi, G.S.: Efficient Detection of All Pointer and Array Access Errors. In: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation. p. 290–301. PLDI '94 (1994)
5. Bao, T., Burket, J., Woo, M., Turner, R., Brumley, D.: BYTEWEIGHT: Learning to recognize functions in binary code. In: 23rd USENIX Security Symposium (USENIX Security 14). pp. 845–860. San Diego, CA (Aug 2014)
6. Caballero, J., Grieco, G., Marron, M., Lin, Z., Urbina, D.: ARTISTE: Automatic generation of hybrid data structure signatures from binary code executions
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. Dhumbumroong, S., Piromsopa, K.: BoundWarden: Thread-enforced spatial memory safety through compile-time transformations. Science of Computer Programming **198**, 102519 (2020)
9. Dhurjati, D., Adve, V.: Backwards-compatible array bounds checking for c with very low overhead. In: Proceedings of the 28th international conference on Software engineering. pp. 162–171. ACM (2006)
10. Dhurjati, D., Kowshik, S., Adve, V.: SAFECode: enforcing alias analysis for weakly typed languages. In: ACM SIGPLAN Notices. vol. 41, pp. 144–157. ACM (2006)
11. dwarfstd.org: Dwarf debugging information format. http://www.dwarfstd.org/doc/DWARF4.pdf (2021)
12. Eliben, p.: pyelftools. https://github.com/eliben/pyelftools (2021)
13. ElWazeer, K., Anand, K., Kotha, A., Smithson, M., Barua, R.: Scalable variable and data type detection in a binary rewriter. SIGPLAN Not. **48**(6), 51–60 (Jun 2013)
14. Hasabnis, N., Misra, A., Sekar, R.: Light-weight bounds checking. In: Proceedings of the Tenth International Symposium on Code Generation and Optimization. pp. 135–144. CGO '12, ACM, New York, NY, USA (2012)
15. Henning, J.L.: Spec cpu2006 benchmark descriptions. SIGARCH Comput. Archit. News **34**(4), 1–17 (Sep 2006)
16. Jim, T., Morrisett, J.G., Grossman, D., Hicks, M.W., Cheney, J., Wang, Y.: Cyclone: A safe dialect of c. In: Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference. pp. 275–288. ATEC '02 (2002)
17. Katz, O., El-Yaniv, R., Yahav, E.: Estimating types in binaries using predictive modeling. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 313–326. POPL '16 (2016)
18. Kratkiewicz, K.: A taxonomy of buffer overflow for evaluating static and dynamic software testing tools. In: In Proceedings of Workshop on Software Security Assurance Tools, Techniques and Metrics. NIST (2006)

19. Lee, J., Avgerinos, T., Brumley, D.: TIE: principled reverse engineering of types in binary programs. In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6-9 February 2011 (2011)
20. Lin, Z., Zhang, X., Xu, D.: Automatic reverse engineering of data structures from binary execution. CERIAS - Purdue University, West Lafayette, IN (2010)
21. Liu, Z., Wang, S.: How far we have come: Testing decompilation correctness of C decompilers. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. p. 475–487. ISSTA 2020 (2020)
22. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: ACM sigplan notices. vol. 40, pp. 190–200 (2005)
23. Maier, A., Gascon, H., Wressnegger, C., Rieck, K.: Typeminer: Recovering types in binary programs using machine learning. In: Perdisci, R., Maurice, C., Giacinto, G., Almgren, M. (eds.) Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 288–308. Springer International Publishing (2019)
24. Matsakis, N.D., Klock, F.S.: The rust language. In: Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology. p. 103–104. HILT '14 (2014)
25. Meng, X., Miller, B.: Binary code is not easy. Proceedings of the 25th International Symposium on Software Testing and Analysis (2016)
26. Metrics, S.S.A., Evaluation, T.: Nist juliet test suite for c/c++. https://samate.nist.gov/SRD/testsuite.php (2010)
27. 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)
28. National Security Agency ghidra, N.: Ghidra. https://www.nsa.gov/resources/everyone/ghidra/ (2019)
29. Necula, G.C., McPeak, S., Weimer, W.: Ccured: Type-safe retrofitting of legacy code. SIGPLAN Not. **37**(1), 128–139 (Jan 2002)
30. Noonan, M., Loginov, A., Cok, D.: Polymorphic type inference for machine code. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 27–41. PLDI '16 (2016)
31. Pang, C., Yu, R., Chen, Y., Koskinen, E., Portokalidis, G., Mao, B., Xu, J.: Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask (2020)
32. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: Addresssanitizer: A fast address sanity checker. In: Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12). pp. 309–318 (2012)
33. Seward, J., Nethercote, N.: Using valgrind to detect undefined value errors with bit-precision. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference. p. 2. ATEC '05 (2005)
34. Simpson, M.S., Barua, R.K.: Memsafe: ensuring the spatial and temporal memory safety of c at runtime. Software: Practice and Experience **43**(1), 93–128 (2013)
35. Slowinska, A., Stancescu, T., Bos, H.: Howard: A dynamic excavator for reverse engineering data structures. In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011. The Internet Society (2011)
36. Slowinska, A., Stancescu, T., Bos, H.: Body armor for binaries: Preventing buffer overflows without recompilation. In: Proceedings of the 2012 USENIX Conference on Annual Technical Conference. p. 11. USENIX ATC'12 (2012)

37. Szekeres, L., Payer, M., Wei, T., Song, D.: Sok: Eternal war in memory. In: Proceedings of the 2013 IEEE Symposium on Security and Privacy. p. 48–62. SP '13 (2013)
38. Wang, H., Xie, X., Lin, S.W., Lin, Y., Li, Y., Qin, S., Liu, Y., Liu, T.: Locating vulnerabilities in binaries via memory layout recovering. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 718–728. ESEC/FSE 2019 (2019)
39. Xu, Z., Wen, C., Qin, S.: Learning types for binaries. In: International Conference on Formal Engineering Methods. pp. 430–446. Springer (2017)
40. Zitser, M., Lippmann, R., Leek, T.: Testing static analysis tools using exploitable buffer overflows from open source code. In: Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering. p. 97–106. SIGSOFT '04/FSE-12 (2004)

Appendix A Optimized Benchmarks

Figure 6 shows the results from the static analysis phase and compares the accuracy of array bounds detection, pointer identification, and instruction owner detection for *optimized* binaries.

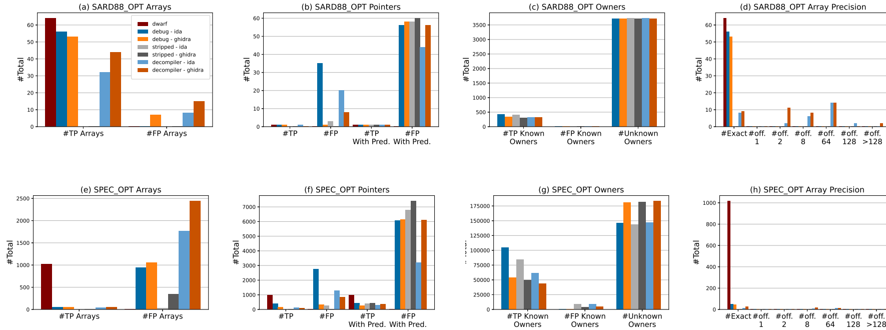


Fig. 6: Accuracy of array, pointers, and owner detection for SARD-88(Optimized), SPEC-cpu2006(Optimized)

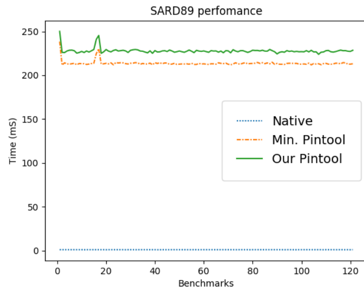
Appendix B Detection Accuracy Using Ghidra

Table 3 shows the detection accuracy of Ghidra for SARD-88 benchmarks.

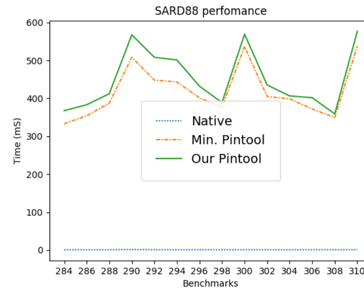
Appendix C Program Execution Time Overhead by the Pin-Based Run-time Technique

Table 3: SARD-88 Test Results (Ghidra) for our three experimental configurations: ① Debug, ② Stripped, and ③ Decomp. (Stripped + Decompiler)

Benchmarks		Debug	Stripped	Decomp.	Benchmarks		Debug	Stripped	Decomp.
I	283	✓	✗	✗	VIII	297	✗	✗	✗
	284	✓	✗	✓		298	✗	✗	✗
II	285	✗	✗	✗	IX	299	✗	✗	✗
	286	✗	✗	✗		300	✗	✗	✗
III	287	✓	✗	✗	X	301	✗	✗	✗
	288	✓	✗	✗		302	✗	✗	✗
IV	289	✓	✗	✓	XI	303	✗	✗	✗
	290	✓	✗	✓		304	✗	✗	✗
V	291	✓	✗	✓	XII	305	✗	✗	✗
	292	✓	✗	✓		306	✓	✗	✗
VI	293	✓	✗	✗	XIII	307	✗	✗	✗
	294	✓	✗	✓		308	✓	✓	✓
VII	295	✓	✗	✓	XIV	309	✗	✗	✗
	296	✓	✓	✓		310	✗	✗	✗



(a) SARD-89



(b) SARD-88

Fig. 7: Program execution time in msec