Memory Safety for Binary Programs

Ruturaj Vaidya\(^1\), Prasad A. Kulkarni\(^1\), and Michael R. Jantz\(^2\)

\(^1\) University of Kansas, Lawrence KS 66045, USA
\(^2\) University of Tennessee, Knoxville TN, USA

Abstract. Security vulnerabilities caused by the illegal use of memory accesses are common. While effective compiler-driven mechanisms to detect and prevent memory errors at run-time are available, comparable binary-level techniques face many challenges. Techniques to ensure that all array references are within bounds at run-time require knowledge of the data types accessed by memory load/store and pointer move instructions, along with precise array bounds. In this work we conduct a detailed study on how accurately can this information be derived using current state-of-the-art static reverse engineering and type inference platforms and for binaries compiled with and without debug symbol information and compiler optimizations. We also develop a novel hybrid and decoupled binary-level technique to detect and prevent illegal memory accesses at run-time. We deploy our technique to study the properties and effectiveness of the information provided by the static reverse engineering 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 of static binary analysis techniques and shows problem areas where further progress is needed to develop accurate binary-level memory safety techniques.

1 Introduction

Memory corruption attacks rely on exploiting illegal memory accesses caused by bugs in software written in low-level memory unsafe languages, like C or C++ \[44\]. Such memory errors in type unsafe languages 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. Many solutions have been proposed to alleviate this security issue. Source-level techniques include the use of memory-safe system languages like Cyclone \[19\] and Rust \[25\], and compiler-level techniques such as CCured \[29\], StackGuard \[4\], SoftBound \[27\], Baggy bounds checking \[1\], and others \[2, 10, 15\]. Binary level techniques have also been proposed and include, Memcheck \[35\], BinArmor \[40\], and others \[46\].

Vulnerability detection and attack mitigation techniques implemented at different level of the system stack have distinct advantages and drawbacks. Source code and compiler level techniques have the best access to program syntax and semantic information to improve precision, completeness, and reduce run-time overhead. But, techniques at this level require access to the source code and are
therefore not applicable to legacy software where the source code may not be available. These techniques may 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. An approach that requires the source code also leaves the task of memory safety solely in the hands of the software developer (rather than the end-user), who typically tend to prioritize performance over security in most application domains. Additionally, the generated binary code may not observe the semantics in the source code [3].

Binary-level techniques have compelling advantages since they do not require the source code and can be applied by the end-users according to their security preferences. Unfortunately, these techniques are generally not as capable since they lack access to high-level program semantic information in binaries stripped of debug symbols. Advanced static or dynamic reverse engineering and type inference techniques to recover the lost debug and semantic information have been developed, but their effectiveness to detect or prevent memory errors has not been thoroughly studied.

Techniques to detect memory errors require the ability to identify memory dereferencing instructions (load/stores) operating over arrays/vectors along with accurate base and bound information of the pointer being referenced. These techniques also need to identify all relevant pointer move instructions to track and propagate data type information. While this information is largely available at the source-code level, much of it may be lost during the compilation process, especially when the generated binary is stripped of debug symbols. In this work we conduct a detailed study on how accurately can this information be derived for binaries compiled with and without debug symbol information. In recent years researchers have developed sophisticated decompilers and binary type inference algorithms that attempt to reconstruct information lost during the source to binary translation process. Our work explored the capabilities of two such state-of-the-art tools, NSA’s Ghidra [41] and Hex-Ray’s IDA Pro [18], to improve the accuracy of the information derived from stripped binaries.  

Most existing techniques to prevent buffer overflows at run-time employ source-code level information, like SoftBound [26, 27], or rely on information gathered from prior dynamic traces of program execution [40]. In this work we develop and assess a new static analysis based binary-level technique to detect and prevent memory errors at run-time. Our approach statically analyzes the input client-side binary (using some reverse engineering framework) to gather information regarding the buffer bounds and the type referenced by each memory access (read/write) and pointer assignment instruction (called the owner) in the binary. This information forms the interface between the static and dynamic components of our framework. At run-time, we employ the Pin [23] virtual machine to keep track of owner information as pointers are assigned and check

---

3 We studied many other reverse engineering tools, including Angr [36, 37, 42], Radare [31], Debin [16], ..., and found that they lack the capability or API for this task.
relevant buffer reads/writes to ensure fine-grained memory safety. We study the
class challenges, effectiveness and efficacy of this binary-level technique to provide
memory safety for client-side program binaries.

In this paper we build our framework and develop experiments to assess the
properties and effectiveness of binary-level memory safety techniques in different
situation. We first attempt to understand cases when binary-level techniques
fail even when the input binary program has access to complete debugging sym-
bol information. We then evaluate cases when the client-side binary is stripped
of symbol information. Finally we evaluate the benefit of two of the most so-
phisticated reverse engineering tools to improve the effectiveness of binary-level
memory safety techniques for stripped binaries.

Thus, we make the following contributions in this work.

1. We conduct a detailed study on the challenges of obtaining the necessary
information for binary-level techniques to detect and prevent memory er-
ers. We assess and describe reasons when even the complete availability of
debug symbol information is not sufficient to replace the relevant semantic
information lost during program translation.
2. We assess whether advanced static reverse engineering and type inference
algorithms can regenerate or predict symbol information with sufficient ac-
curacy to improve effectiveness of memory protection techniques for stripped
binaries.
3. We develop and describe a novel decoupled binary-level techniques for com-
plete memory safety for binary programs, and employ it to assess efficacy,
challenges, and performance of binary-level memory safety techniques in dif-
ferent situations.

The remaining paper is organized as follows. We present additional back-
ground about the memory safety and the frameworks we employ for this work in
Section 2. We describe our experimentation framework and benchmarks in Sec-
tion ?? We describe the capability of reverse engineering frameworks to extract
program information in Section 4 We describe our novel binary level memory
protection technique in Section 5. We describe our experimentation framework
and benchmarks in Section ?? We present our experiments and observations in
Section 6. We present related works in Section ?? Finally, we present avenues
for future work and describe our conclusions in Sections 7 and 8 respectively.

2 Background and Related Works

Languages like C/C++ provide the programmer with a concept of pointers that
allow unrestricted arithmetic and that are not bound to a single unique memory
object (sometimes called its “intended referent” in prior work [11,20] and “owner”
in this work). Yet, the ANSI language specification establishes some rules that
guide the correct and expected pointer behavior. Thus, although pointer ma-
nipulation within objects is allowed, pointers are not permitted to cross object
boundaries. Contrariwise, the layout of objects in memory is not defined by the
specification and can not be assumed by programmers. There is a long history of work in detecting and preventing memory errors in type-unsafe languages. In this section we provide an overview of memory errors and existing approaches and related works to guard against such errors.

Memory errors are malicious attacks that utilize the process memory to change program behavior. Languages like C and C++ lack type safety and memory safety, which leads to memory corruption errors like stack-based buffer overflow [5], heap-based buffer overflow [6], double free [7], use after free [8], etc. Memory errors are employed by attackers to change the flow of code and direct it to the attacker specified location, such as shell code, either directly or using gadgets (small instruction chunks) present in the memory to craft malicious attacks.

Memory errors can be categorized into two types [43]: (a) spatial errors allow the attacker to overflow memory buffers, while (b) temporal errors allow illegal memory use by dereferencing invalid pointers. Figure 1 shows an example of a spatial memory safety violation, where memory accesses overflow the legal bounds of the buffer pointed to by ‘p’. Figure 2 shows an example of a temporal memory safety violation, where a pointer is accessed after the memory buffer it points to has been freed. Both spatial and temporal memory safety is important to ensure the program safety. Our current work aims to prevent spatial memory errors, and therefore we focus our remaining discussion in this section on issues and techniques related to such errors.

```
1 char *p = malloc(10);
2 for (int i = 0; i < 15; ++i)
3     // BAD
4     *(p + i) = i + 65;
```

Fig. 1: Spatial Memory Safety Violation

```
1 char *p = malloc(10);
2 free(p);
3 for (int i = 0; i < 10; ++i)
4     // BAD
5     *(p + i) = i + 65;
```

Fig. 2: Temporal Memory Safety Violation

Traditionally, approaches to prevent (spatial) memory errors fall into two main categories: (a) pointer-based techniques that employ a fat pointer, where the object base and bound information is associated with each memory pointer [27, 29, 38], and (b) object-based techniques that associate and track base and bound information with each object [1, 9, 12, 34]. Pointer-based techniques change the program’s pointer representation and memory layout that cause significant com-
compatibility issues when interfacing code that is using such techniques with library
code not using fat pointers. Pointer-based techniques also need to propagate and
check bounds for all pointers, which can cause significant performance overhead.
In contrast, object-based techniques maintain object information in a separate
table. They do not update the pointer representation and do not cause any compati-
bility issues. However, object-based techniques have difficulty distinguish-
ing between two pointers that hold the same memory address, but one that
points to the whole structure and the other that points to only the first sub-
field within that structure. Object-based techniques need to frequently perform
a range lookup to map the pointer address to the correct object in the external
table, which can also cause high performance overhead.

In recent years, both the pointer-based and object-based approaches have
developed new techniques to overcome the traditional drawbacks of each ap-
proach. Thus, newer pointer-based techniques have resolved their compatibility
issues by employing a separate table to hold pointer related metadata [27]. Like-
wise, newer object-based techniques have also resolved many of their sub-object
overflow issues [12]. Most of these past approaches work on the source-code or
intermediate-code level. In the rest of this section we compare our work with
binary-level techniques that are related to our work.

Tools such as Purify [33] and Valgrind Memcheck [30] can protect binary
programs against memory corruption. These tools shadow every byte of memory
to detect undefined and out-of-bounds memory accesses. They require debugging
symbol tables, cannot locate buffer overflow within the stack and global memory
regions, and are designed to be employed as memory debuggers rather than online
during production runs due to the orders of magnitude overhead they typically
impose.

Software-based fault isolation (SFI) techniques, such as XFI [14] also con-
strain memory reads and writes to only access the data region within its fault
domain. However, SFI has a distinct goal of isolating untrusted components into
separate protection domains (rather than providing memory safety) and such
techniques also do not support fine-grained memory protection [45].

Especially related are binary-level techniques that researchers have developed
to locate fine-grained buffer overflows in memory. The BinArmor technique [40]
to detect memory errors relies on a tool called Howard [39] that uses past pro-
gram execution traces to extract data structures and their memory bounds and
find potentially unsafe pointer accesses to the detected buffers. BinArmor uses
the information from Howard to statically instrument the binary with additional
checks to detect unsafe memory accesses during later program execution. An-
other recent technique develops an enhanced memory layout recovery algorithm
to locate memory access vulnerabilities in the program after execution of the
failed run [46]. 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 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. We also develop a novel hybrid implementation where the detection mechanism is more clearly separated from the analysis algorithm via a simple and readable interface. 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.

3 Benchmarks and Frameworks

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

3.1 Benchmarks

In this work we use benchmarks from three different benchmark suites, SARD-89 [7,21], SARD-88 [?,47], and SPEC cpu2006 [17]. The SARD-89 suite contains 291 small subjects that implement a taxonomy of 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 the technique of 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 \texttt{buf[2048]}. We discuss these cases in more detail in Section 4.2. We use the remaining 121 test subjects for all experiments in this work.

The SARD-88 benchmark suite contains 14 model subjects from various "Real-World" 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 linked important library functions like \texttt{strcpy}, \texttt{strcmp}, etc. from \texttt{string.h} library to subsist complete buffer propagation through program execution. We also employ 7 programs from the SPEC cpu2006 integer benchmarks to evaluate the scalability, performance and efficacy of the static and dynamic tools on large programs. Note that all test cases we consider only consist of C programs.
3.2 Experimental Configuration

All the experiments are performed on a x86 64-bit Intel Xeon processor. Fedora 28 is used as an operating system with kernel version 5.0.9-100.fc28.x86_64. We use Ghidra version 9.1.2 and IDA Pro version 7.5 (with Hex-Rays decompiler) to conduct static analysis. Scripts for Ghidra are written using Python 2.7, while Python version 3.6 is used for IDA Pro scripts. Our Pintool scripts are compiled using the GCC/G++ compiler suite version 9.3.1 and PIN version pin-3.15-98253-b56e429b1 is leveraged to perform run-time experiments.

3.3 Tools and Frameworks

Static Analysis and Reverse Engineering Frameworks Static binary analysis frameworks employ sophisticated reverse engineering algorithms to discover properties of the input binary code, including the detection of function boundaries, distinguishing code from data, constructing control-flow graphs and callgraphs, finding targets of indirect jumps, and inferring types. Several frameworks provide capabilities like debugger, code diffing, binary rewriting, etc. Some frameworks also include a decompiler that attempts to reconstruct the high-level source code representation of the given binary code. In this work we employ the Ghidra framework [28] developed and open-sourced by the United States National Security Agency in 2019 and IDA Pro [32] commercial disassembly framework developed by Hex-Rays.

Data Type Reconstruction using Ghidra and Hex-Rays Decompiler A decompiler’s job is to convert binary code to a high level language representation like C. It employs various techniques to obtain high level information like data types, structures and aggregates from stripped binary. In this work we employ two decompilers, one provided with Ghidra and the other by Hex-Rays with IDA-Pro (version 7.5), to recover local data structure information from stripped binaries. The IDA Pro reverse engineering tools are often considered the de facto industry standard. The two decompilers reconstruct lost type information in stripped binaries. We employ the respective APIs and interfaces to output the reconstructed type information by Ghidra and Hex-Rays decompilers. This information is read back and applied by the disassembler interfaces in lieu of debug symbol information in stripped binaries. Thus, we assess the quality of the static data type recovery algorithms in these two decompilers and their ability to improve the effectiveness of our technique to detect memory errors for stripped binaries.

Dynamic Instrumentation Frameworks Dynamic instrumentation engines are powerful tools that employ various techniques to control program execution and insert new or update existing instructions at user-defined instrumentation points to record program behavior or alter program execution. In this work, we use the popular Pin dynamic instrumentation tool released by Intel [23].
4 Static Reverse Engineering

Techniques to detect and prevent memory errors need precise information regarding vector/buffer data types, their base address and size/bound, and where and how they are used in the program. While this information can be more readily identified from the program source code, much of it is lost during the binary code generation process. Reverse engineering frameworks employ complex algorithms and heuristics to regenerate this lost program information from binaries. In this section we explore the abilities of two sophisticated reverse engineering tools to identify and reconstruct program information that is required to detect and prevent memory-related attacks in binaries.

4.1 Setup and Implementation Details

In this section we describe our algorithms, techniques and extensions developed to explore the capabilities of the reverse engineering frameworks used in this study, NSA’s Ghidra and Hex-Ray’s IDA Pro. Many reverse engineering frameworks, including the ones used in our study, provide a rich user-level API to write scripts to extract information gathered by the tool. We have developed scripts in each framework to extract information relating to the statically known buffer bounds (local and global variables) and owners for all memory access instructions. Owners are the program variables that are referenced by the load/store and pointer move instructions in the binary code (called the intended referent in some earlier works [20] for instructions that use pointers). 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.

We use Figure 3 to illustrate the information we gather from the static reverse engineering tools. The figure shows the source code, the compiler generated binary code and corresponding IDA Pro output for a simple C program. This source code representation of the program has a single integer buffer, ‘b’, an integer pointer, ‘ptr’, and an integer scalar ‘n’. This pointer access on line #6 overflows the array ‘b’. Memory safety algorithms need to check such accesses to determine the invalid access at run-time.

In the binary representation of the program, the assembly instructions at offsets ‘8’ and ‘20’ denote direct pointer ‘ptr’ access; on the contrary, instruction at offset ‘27’ is equivalent to pointer ‘ptr’ dereference. Instruction at offset ‘4’ represent buffer ‘b’ access, while instruction at offset ‘c’ denote direct access of integral variable ‘n’. We determine ‘ptr’, ‘b’ and ‘n’ as “instruction owners” of the corresponding addresses. The assembly instructions at offsets ‘4’ and ‘8’ indicate pointer assignment, while instructions at offsets ‘13’ to ‘27’ represent source code *(ptr+n) = 4;*, indicating the memory access violation. Notice that the pointer ‘ptr’ is being moved to the register rax at offset ‘20’ and invalid memory access occurs at offset ‘27’.

The owners of direct variable access instructions (instructions which employ \{rbp, rsp, rip\} relative addressing) are generally determined automatically by
reverse engineering frameworks, conversely, the *owners* of pointer dereference instructions may not be detected automatically even by advanced tools, including IDA Pro. To catch such illegal memory accesses, we have implemented an analysis algorithm that keeps track of the registers where the pointer address is first copied. In this case, the offset ‘20’ holds the pointer location value. Our data-flow algorithm can track such pointer movements even in more complex situations. Then, it checks for all the memory accesses using such registers that point to memory regions. Thus, in this case, the *owner* of the instruction at offset ‘27’ is determined as the pointer variable ‘ptr’.

Figure 3(c) shows the output of our reverse engineering scripts after analysing the binary compiled using example program shown in Figure 3(a). This output file contains addresses mapped to their “instruction owners” (listed under *addresses*), function local variables (listed under *locals*) and global variables (listed under *.global*). The program instructions and addresses listed under the *addresses* field determine the instructions to be checked for overflow at run-time. Note that only "MOV" instructions are required to be instrumented, and only arrays and pointer type objects are considered for our analysis; consequently, only such instructions correspond to pointer and array types are included in final output. A careful reader may notice that the binary is compiled by retaining debug information, as evident by correct array size deduction despite of inadequate context information.

The reverse engineering tools determine the variable names using debugging (symbol) information, or they assign names if the symbol information is not available. Local variables are represented in the output file as *function-name_variable* along with their position (offset) on the stack relative to the stack pointer, their size and data type. Variables defined in the *data or bss* section are represented as *.global_variable* and associated with their static address; the rest of the metadata is similar to local variables. The static tools also provide metadata such as number of functions, function boundary, stack size, base pointer relative addressing information and additional metadata reserved.
for future implementation. The output of this static analysis is fed to the Pin tool.

### 4.2 Efficacy of Reverse Engineering Tools

To correctly detect and prevent memory errors, we need precise information from the static analysis tools regarding the base address and size of every vector data type, and the owner or intended referent of each memory access instruction in the program. In this section we present our experimental results that study the efficacy of existing reverse engineering tools to determine this information for programs compiled by standard compilers with and without debug symbol information and compiler optimizations. In particular, we explore the following questions.

1. Even in the case when complete debugging symbols are retained, does the translation process from high-level code to binary result in a loss of program semantic program information that can make it infeasible for static tools to gather precise program information?
2. How effectively can reverse engineering tools analyze binaries that are stripped of all debugging information?
3. How effectively can recent state-of-the-art reverse engineering and type inference algorithms regenerate lost program information in stripped binaries?
4. What is the impact of compiler optimizations on the efficacy of static reverse engineering?

---

```
#include <stdio.h>

int main ()
{
    int b1[5];
    int b2[10];
    b2[15] = 1;
    printf("%i\n", b1[3]) ;
    return 0;
}
```

**Fig. 4:** Ambiguous array access: (a) C source code (b) Assembly output

---

**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 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 4. 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]’. Then, the write to ‘b2[15]’ corresponds to assembly instructions 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. If these accesses are within the stack frame, then from the assembly code it is hard to infer or predict if they refer to the array ‘b1’ or ‘b2’ or neither. In such cases, our reverse engineering tools cannot determine the correct instruction owner, albeit in the presence of debug symbols.

Analysis of the above example is even more complex because ‘b2[15]’ and ‘b1[3]’ map to the same location in the stack memory. This ambiguity makes it impossible to distinguish between the two accesses, one of which is correct while the other one leads to an overflow (or is undefined). Such failures caused due to buffer accesses by a constant numeral are an intrinsic limitation of binary-level techniques. Fortunately, arrays dereferenced by a constant numeral may not be a security hazard or attack vector in many security threat models. A similar issue arises due to the lack of owner information encoding in binaries to accurately detect structure member access instructions correctly in a few cases. There are 152 test cases in the SARD-89 benchmark suite that fail due to these reasons. We leave out these subjects from the remaining experiments in this paper.

**Accuracy of Type Detection for Arrays and Pointers** Figure 5 display the efficacy of array and pointer type detection for programs in the SARD-89, SARD-88, and SPEC suites, respectively. The programs are compiled in non-optimized as well as optimized settings. Each figure shows three configurations for each of our two static reverse engineering tools, one for binaries compiled with debug symbols, second from binaries stripped of all symbols, and the third for stripped binaries with information derived from each tools’ decompiler/type inference algorithms that attempt to reconstruct the high-level program information lost in (stripped) binaries. Information retrieved from binaries compiled with debug symbols is considered a baseline and is used to compare the data inferred in the other two configurations. We leverage *pyelftools* [13] module to extract this information from ”dwarf” section of binaries.

Sub-figures {a, e, i, m, q} in Figure 5 represent stack array detection accuracy for corresponding benchmarks. #TP Arrays belong to the arrays detected at correct offsets regardless of their size, while #FP Arrays belong to the arrays detected at incorrect offsets or those which are incorrectly estimated. Both IDA Pro and Ghidra manifest very good accuracy in presence of debug symbols on smaller (SARD 89) benchmarks, however Ida Pro outperforms Ghidra on bigger benchmarks. Unsurprisingly, array detection accuracy fall in proximity to zero in stripped benchmarks. The reason is the higher level structure recovery occurs during decompilation. Moreover, few instances of ”correct arrays” are false positives (todo:). Ghidra decompiler slightly surpasses IDA Pro’s Hey
Fig. 5: Accuracy of Stack Arrays and Pointers detection for SARD-89, SARD-88, SARD-88(Optimized), SPEC-cpu2006, SPEC-cpu2006(Optimized). The maroon bar denotes actual types present on stack

Rays decompiler in recovery of lost arrays at correct offsets (todo: about correct bounds?).

It is also crucial to discuss the accuracy of array bounds detection along with their position on stack. Figures 5{d, h, i, p, t} depict array size detection accuracy of #TP Arrays and how they differ compared to the actual bounds. Despite having much higher array detection accuracy in terms of their stack position, the figures indicate that the accuracy of array size detection degrades especially in the absence of debug symbols.

Sub-figures {b, f, j, n, r} in Figure 5 constitute pointer detection accuracy of reverse engineering frameworks. The first set of graphs in each figure, represent
Memory Safety for Binary Programs

# TP Pointers and FP Pointers
without prediction and the second set represents those with prediction. The pointers without prediction are determined "as-is" by reverse engineering tools while pointers with prediction are marked as pointers by our prediction algorithm. Currently we mark every variable with "undefined type" (or undetermined type) and "size 8" as a pointer, nonetheless a sophisticated algorithm could be employed to track pointer type during data flow to improve the prediction. The accuracy with prediction is higher than that without prediction, though, it produces more false positives.

Accuracy of Owner Detection for Memory Access and Pointer Move Instructions
The accuracy of owner detection for pointer and array access instructions is presented in sub-figures \{c, g, k, o, s\} of Figure 5. The owners are divided into two categories - known owners and unknown owners. The known owners are the owners which get predicted by our stack analysis algorithm and associated to instructions to be instrumented, while unknown owners denote "unowned" instructions or when relevant owners to the instructions can’t be predicted. All results are compared against benchmarks when debug symbols are available. The instructions are assigned with very high number of unknowns, chiefly in optimized benchmarks. The primary reason is that our tool currently does not consider register allocated variables.

5 Binary-Level Technique to Detect Memory Errors

In this section we describe the implementation details of our novel binary-level technique that employs a hybrid static analysis and dynamic tracking and instrumentation approach. We adopt a decoupled hybrid approach to take advantage of the contrasting strengths and weaknesses of static and dynamic approaches with regards to the following properties.

5.1 Protection from Memory Errors
PK: Talk more generally about approaches to protect from memory errors. Can move from here and integrate with section that describes our technique.

In this work we develop a pointer-based approach to guard against spatial memory errors that works on program binaries and does not require access to source code or hardware support. Our hybrid approach extracts the needed information regarding pointers, array bounds, and instructions de-referencing pointers from static reverse engineering tools, which is then used at run-time to detect and prevent memory errors. The run-time implementation stores and propagates the base and bound metadata associated with pointers in a disjoint metadata structure. On every pointer load and store, instrumentation inserted by the run-time system inspects whether the pointer accesses are within the buffer bounds.

Figure 6 lists a simple C program to illustrate the checks inserted to detect memory errors in a simple case. Here, size is the size associated with the object
1. **Analysis capabilities:** Static binary analyzers have a whole-program view and can afford to spend longer times to find the desired program information. In contrast, dynamic binary analyzers can only view information one block or one trace at a time. While dynamic analyzers can store state gathered from previously seen code, they do not typically analyze code not reached during execution. They are also constrained on time, since any time spent doing analysis can potentially slow down the program execution.

2. **Code coverage:** Static analyzers may not be able to discover all the code due to the combination of hard code constructs, such as indirect jumps and calls, data interspersed with code, padding bytes, and variable-sized instructions. These issues place limits on the accuracy of current code discovery and CFG detection algorithms. Shared library code (on client systems) can often not be analyzed. Even when library code could be analyzed, instrumenting shared library code is problematic. In contrast, dynamic rewriters can see and instrument any code that is reached during execution, including library code.

3. **Flexibility:** A binary that is statically rewritten cannot be easily adapted to different security policies, say ranging from no security to the most heightened level of security. Software developers will need to construct a separate binary for each case. In contrast, dynamic rewriting can be configured with a different security policy during each run, or the policy can even change during the run.

Our hybrid implementation inspired by the SoftBound technique uses Ghidra for static analysis and Pin for dynamic analysis and instrumentation.

A schematic of our approach is illustrated in Figure 7. We employ modular approach to provide flexibility in tool replacement. The operation occurs in two phases - static analysis phase and DBI (dynamic instrumentation) phase. The input binary is first statically analyzed to recover following properties about the binary program: (a) function and instruction disassembly, (b) local variable and data type information, (c) *instruction owner* mapping, and (d) high order data structure and array bound recovery. This information is extracted from Ida Pro and Ghidra using their standard API and written into an interface file. The Pin runtime engine executes the given binary. The program input and the statically generated interface file are provided to Pin. Pin adds instrumentation based on previously determined *owner* mapping, tracks dynamically allocated buffers and
relevant register and memory values, and inserts security checks to detect buffer overflows. The following sections provide further details about the approach.

5.2 Dynamic Tracking and Instrumentation using Pin

Our hybrid approach relies on a dynamic instrumentation engine to utilize the information supplied by the static reverse engineering tool in the interface file to detect the actual memory safety violation, or buffer overflow errors, at runtime. Specifically, our current implementation uses the dynamic runtime engine, Pin [23]. We build scripts, called Pintools, that use the Pin API to insert dynamic checks in the executed code. Algorithm 1 represents our dynamic overflow detection algorithm.

The interface file provides information such as function start $f_s$ and function end function end $f_e$ (function boundary), function metadata $f_m$ - which includes function stack size, variable offsets and bounds, etc., instruction-owner mapping $I$. It also provides Global metadata which includes global variable information such as location and bounds. The Pintool collects and stores the function and global metadata in a global map structure. Pintool also stores the instruction and owner mapping in a separate structure. This structure maps instruction address as a key to an object containing the corresponding instruction owner metadata as a value, and makes it easier and quicker to access the instruction owner and other metadata during instruction instrumentation. The runtime information per function is stored in a stack structure, which gets updated each time a function gets called or returned.

The instruction instrumentation API provided by Pin is used to implement the detection mechanism. Instruction instrumentation is required to add instrumentation for instructions which are mapped with corresponding instruction
Algorithm 1: Object overflow detection

Input: Function Metadata \( F \rightarrow \sum f_i \{(f_s \cap f_e), f_m, I\} \)
Input: Global Metadata \( \{Gm\} \)

1. ReadInput(Input);
2. InstrumentMallocFree();
3. \( F_d \rightarrow \) Set of functions reached during execution;
4. \( I_d \rightarrow \) Instructions per function reached during execution;
5. foreach \( f \in F_d \) do
6.     foreach \( i \in I_d \) do
7.         if \( i.Address == f_s \) then InitializeStack();
8.         if \( i.Address == f_e \) then UnInitializeStack();
9.         if \( i.Owner \in Unknown \) then Continue ;
10.        if IsMemStore(i) then
11.            if \( i.Owner \in Pointer \) then
12.                if \( i.BaseReg \subseteq \{rbp, rsp, rip\} \) then
13.                    BoundPropogationCheck(i.Owner, i);
14.                else PtrBoundCheck(i.Owner, i);
15.            else ObjBoundCheck(i.Owner, i);
16.        end
17.    else if IsMemLoad(i) then
18.        if \( i.Owner \in Pointer \) then PtrBoundCheck(i.Owner, i);
19.        else ObjBoundCheck(i.Owner, i);
20.    end
21. end
22. end
owners as shown in Figure 3 (c). We statically identify following instrumentation categories based on which run-time checks get added.

I. Function start - stack initialization

Instruction instrumentation is added at each function start, to dynamically decide locations of local variables w.r.t. the actual value of stack pointer in memory. function start ($f_s$) address obtained from static pass helps in determining instrumentation point. The actual variable locations i.e. associated bounds get stored in a global metadata structure in this phase. Program arguments are also detected in this phase by adding a special check for function ‘main’.

II. Function end/return - stack roll-back

Similar to function start, function end ($f_e$) is leveraged to calculate the run-time location of function end address. This type of instrumentation is required to roll-back the allocated stack and remove corresponding variables.

III. Pointer move - propagate (buffer base, bound) info.

Pointer assignment instructions are used to allocate address of buffer to a pointer, so that pointer can be used to indirectly access the buffer. Consequently, it is important to transfer and associate buffer bounds to that pointer. Consider following instruction pattern discussed in Figure 3.

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

Here, `lea` (load effective address) instruction brings address of a buffer into a register, and further it assigns it to the pointer. Thus, static analysis tools mark owner of instruction at offset ‘8’ as pointer `ptr`. A run-time instrumentation is added to check the contents of register, and the location of object $b$ in memory is determined. Note that the address and bounds of $b$ get stored in a global map structure during stack initialization. These bounds are then transferred to the pointer. Note that an immediate is often used to access global objects. As, absolute address of global objects is known during compilation. It is depicted in following instruction pattern.

```
lea rax,[rbp -0x10],0x402010
```

Contents of section .rodata:

```
402010 "string"
```

IV. Pointer dereference - check if pointer access is within bounds

Consider following instruction pattern from Figure 3.

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

The instruction at offset ‘27’ represents pointer dereference. The pointer contains buffer $b$ bounds and thus it is important to check whether the access is within the bounds. Therefore, instrumentation is added just before this instruction to verify the access as follows.
V. Array/Object bound - check if array access is within bounds

Similar to the pointer dereference check, it is also important to check whether the actual object access is within the bounds.

Apart from the above instrumentation categories, we also instrument dynamic memory allocation functions like `malloc`, `calloc`, etc. We use routine instrumentation to support such dynamic allocation functions. Consider following instruction pattern depicting memory allocation using `malloc` function call.

```c
void foo(int *ptr)
{
    int x = ptr;
    int y = *x + 5;
}

int main()
{
    int b[5];
    b[0] = 1;
    foo(b);
    return 0;
}
```

The instrumentation is added for the malloc function call, which detects the allocated size and the return address. Then it stores this in a map structure. The return address gets stored in `rax` and gets assigned to the pointer `ptr`. Consequently, pointer `ptr` acquires the bounds once the instruction is reached.

Our implementation supports pointer metadata propagation through function calls, i.e. it propagates the pointer bounds information whenever pointers are passed between different functions. Figure 8 presents the bounds propagation technique. Function `main` passes array ‘b’ to function `foo` through register `rdi` (instruction at offset ‘4004c6’). Subsequently, in function `foo`, the value of register `rdi` (which is nothing but the location of array ‘b’) is assigned to a pointer ‘ptr’ (‘4004a9’) (Note that, according to x86-64 - linux calling convention first six arguments are passed through registers and rest of them are pushed on the stack). Hence, pointer ‘ptr’ acquires the bounds of array ‘b’. Next, pointer ‘ptr’ is assigned to pointer ‘x’ (‘4004a2’) and ‘x’ acquires the bounds of pointer ‘ptr’, consequently acquiring the bounds of array ‘b’. Pointer ‘x’ is then used to overflow the assigned object (‘4004aa’).

**Fig. 8:** Bounds propagation: (a) C source code and Assembly output of function `main` (b) and `foo` (c)
6 Experimental Results and Observations

In this section we first discuss the effectiveness of our tool, and then describe the experiments we designed to study the properties, potential and limitations of static reverse engineering based techniques to detect and prevent memory errors in program binaries. We present and describe the results and observations from our experiments.

6.1 Detection Accuracy of Our Tool

We check the effectiveness of our tool on various decompilers using two sets of benchmarks - SARD 89 and SARD 88. SARD 89 benchmarks contains various small-scale cases which facilitate testing our tool with diverse buffer overflow patterns. SARD 88 benchmark set consists of bigger test programs which suffice real world evaluation. We check the effectiveness of our tool in three arrangements - 1) binary programs with debug symbol information 2) stripped binary programs without decompiler 3) stripped binary programs with decompiler. The decompiler phase is required to recover higher order structures like arrays.

Observations with SARD 89 benchmark suite Table 1 presents the efficacy of our tool on SARD 89 benchmark suit with IDA Pro and Ghidra static analysis frameworks. We use two types of special extensions to our tool to improve the accuracy, and discuss efficacy of our tool using three different types of instances.

- **Instance I** In this instance, no special extension is used to produce the results.
- **Instance II** We develop an extension which is only required in Ghidra script to detect certain types of instructions. Consider following pattern which represents array access instruction.

\[ 40112b: \text{mov BYTE PTR } [\text{rbp} + \text{rax} * 1 - 0x16] , 0x41 \]

We observe that the variable reference of such instruction is not assigned by Ghidra's default algorithm. Thus, we use an extended algorithm to predict such cases. We call this **ext-I**

- **Instance III** If our static analysis frameworks fail to associate owner to an instruction, then it is marked as **unknown**. A special extension is applied to our PIN framework to detect overflows in such cases. If owner of an instruction is mapped to **unknown** owner, then we check if array or pointer access is within the bounds of stack. We refer this as **ext-II**

Figure 1{a and b} show efficacy in 121 test cases with non-constant array access. Figure shows that efficiency of Ghidra script increase in **instance-II** where **ext-I** is used, while IDA Pro automatically gives better results without extension. In **instance-II** the efficacy of Ghidra script improved, however IDA Pro results don't get enhanced, especially in stripped case without decompiler, as compared to Ghidra. The reason for that is the array access instruction owners are typed
as "scalars" and are thus ignored by our Pintool. Figure 1{c and d} show efficacy in 152 constant array access test programs. The figure depicts improved efficacy in both IDA Pro and Ghidra settings in instance-III. Such cases are detected as array access instruction owners are typed as "pointers" and thus instrumented by Pintool.

**Observations with SARD 88 benchmark suite** Figure ?? displays SARD 88 benchmark results in IDA Pro and Ghidra settings. Only instance-III is considered while producing the results. Both IDA Pro and Ghidra scripts show better results when debug symbols are available. The detection accuracy is very low in stripped case without decompiler support, and these results get improved with the introduction of type recovery algorithms using decompilers. The overall results show that IDA Pro produces better results than Ghidra. We now discuss the factors which affect the detection accuracy.

- One of the important distinction between IDA Pro and Ghidra is that, IDA Pro almost accurately detects null terminated "printable" strings, which are usually defined in ".rodata" section. This is the reason of failure in benchmarks II, IX, XI and XIV using Ghidra. We also observe that there is a lower bound on how these strings are detected, i.e. strings with length less than 4 are not detected by IDA Pro by default, and Ghidra limits this length to 5 by default. This is a fundamental limitation of reverse engineering tools, as reducing this lower bound on certain architectures may produce false positives.

- Benchmark VIII - The primary reason why this benchmark fails is because of deficiency of reverse engineering tools to detect global variables. The 298 fails because of incorrect global array size detection. An array "buf" (with size 31) in file "crackaddr-bad.c" isn’t correctly detected. On the contrary the same array is correctly detected in case 297, which contains an overflow.

- Benchmark XIII - This benchmark also fails because of the shortcoming of reverse engineering tools in detecting the global array. In case of 307, the benchmark fails as our tool is able to detect overflow due to lack of array notion. On the other hand, case 298 passes because there’s no overflow to begin with.

### 6.2 Instrumentation Categories

Figure 9 show the comparison between static and dynamic instrumentation categories. The static instrumentation categories Figure 9-{a,b} are determined in the static reverse engineering phase. These categories represent pointer move, pointer dereference, array access (Categories III, IV and V as explained in section 5.2) and category unknown. Figure 9-{c,d} represent the dynamic instrumentation categories. In the addition to the explanation in section 5.2, the figures also include the category unknown.
Table 1: SARD-89 Test Results for three experimental configurations: (a) Debug - binaries compiled with debug symbol information, (b) Stripped - binaries with debug information stripped out, (c) Stripped (Ghidra Decompiler) - Originally stripped binaries processed by Ghidra’s decompiler and type inference algorithm.
Table 2: SARD-88 Test Results for three experimental configurations: (a) Debug - binaries compiled with debug symbol information, (b) Stripped - binaries with debug information stripped out, (c) Stripped (Ghidra Decompiler) - Originally stripped binaries processed by Ghidra’s decompiler and type inference algorithm.
Figure 9-(a,c) resonate with accuracy results shown in 1 in both IDA Pro and Ghidra configuration. These graphs reflect high accuracy in Ghidra stripped case, which is manifested by dominant unknown bar in static as well as dynamic instrumentation graphs.

Fig. 9: Instrumentation Categories in SARD89 and SARD88 benchmark settings - (a) and (b) represent instrumentation categories determined statically, (c) and (d) represent instrumentation categories reached dynamically.

6.3 Performance Overhead

7 Future Work

There are several avenues for future work. First, we will work to improve the robustness of our tool and validate these results using other diverse benchmarks. We also plan to develop techniques to further reduce the overhead of the execution-time component of our approach. Second, we will further study and compare the impact of using advanced static type inference and recovery techniques, such as TIE [22] and TypeMiner [24], with dynamic techniques, such as Howard [39], to resolve this issue of detection and prevention of illegal memory accesses at run-time. Third, we will explore if the different static and dynamic type inference approaches can provide complementary detection capabilities that can be combined to enhance the overall precision of the recovered program semantic information. Finally, we plan to pursue our overall goal to develop a suite of binary-level techniques to protect client-side binary programs from different classes of run-time security issues and different run-time overheads and compare their protection capabilities with equivalent source code level techniques.
8 Conclusions

In this work we developed and assessed the properties, capabilities and limitations of the first binary-level static-analysis based technique to detect memory errors during program execution. The observations from our experiments show that, (a) binary-level security mechanisms are likely to face some intrinsic limitations due to loss of program semantic information even when complete debugging symbols are available, (b) as is widely accepted, stripping binaries of debug information can significantly hurt the effectiveness of binary-level security techniques, and (c) binary-level static analysis and type inference algorithms now possess advanced capabilities that can enable reliant security techniques to achieve their goals in many cases even when the binary programs are completely stripped of debugging symbol information. We conclude that our observations are promising but research is needed to further improve the capabilities of binary analysis, perhaps by combining multiple different static and dynamic algorithms. Likewise, binary-level mechanisms to effectively reduce or remove security vulnerabilities from client-side software seem feasible, but further research is needed to improve their effectiveness and reliability, to reduce their run-time overhead, and enhance ease of use to facilitate more general deployment. (d) IN stripped benign case the bound detection is incorrect, because the instruction that contributes is getting detected. checks are added in sard88 in benign cases and thus these cases are failing.

References


