VAPOREON: Validating Accesses to Pointer References during Optimization

Ammar Ahmed
ammarah@umich.edu

Marwa Houalla
mhoualla@umich.edu

Daniel Liu
danlliu@umich.edu

Wenzhao Qiu
qwzhao@umich.edu

***Abstract*—Memory corruption vulnerabilities consistently prove challenging to eliminate and represent a significant portion of modern security vulnerabilities [1]. It is crucial for developers across all levels of the software stack to understand how memory corruption can occur. This often happens through out-of-bounds array accesses, particularly in lower-level languages like C and C++, which do not automatically check bounds. High-profile cases such as Heartbleed[1] and various web browser vulnerabilities[2] underscore the urgent need for better early detection tools.**

We introduce VAPOREON (Validating Accesses to POinter REferences during OptimizatioN), an optimized runtime bounds checker specifically designed for stack objects. Our focus on stack allocations stems from the critical security risks associated with out-of-bounds stack accesses, such as hijacking the return address. By thwarting out-of-bounds writes to these objects, VAPOREON seeks to reduce the attack surface of compiled programs, bolstering them against memory corruption attacks.

I. Introduction

According to the 2024 Google Security Engineering Technical Report, memory safety bugs account for approximately ~70% of severe vulnerabilities in large C/C++ codebases [1]. Despite considerable efforts over the last fifty years to address the challenges of memory-unsafe languages, the need for scalable solutions to enhance memory safety remains urgent. We focus primarily on Spatial Safety,[3] a subset of memory safety concerns that encompasses vulnerabilities such as buffer overflow and out-of-bounds access.

To safeguard programs against these vulnerabilities, implementing bounds checks is essential. This involves maintaining a record of the valid address boundaries for objects, buffers, and arrays. By systematically checking every load and store operation that accesses these data structures, we can ensure that they do not exceed the established boundaries, thereby preventing unauthorized access and potential exploits.

One such example of an exploit that can be prevented with bounds checking is a buffer overflow attack known as "Stack Smashing" [2]. This type of attack occurs when an application, without checking input size, copies external data into a fixed-size buffer on the stack—often using unsafe functions like strcpy. Attackers exploit this by inserting malicious data to overwrite the stack's return address, causing the program to execute the malicious code upon function return. Notable examples of this attack include the Code Red, Nimda, and Slammer worms.

However, despite enhancing security, introducing bounds checking to an application comes with a fairly significant tradeoff. Storing and checking the bounds for each data structure on the stack substantially increases performance overhead. Consequently, a considerable amount of research on spatial safety focuses on developing efficient methods for implementing bounds checking. One such method developed by W. Chuang, S. Narayanasamy, B. Calder, and R. Jhala [3] investigates the possibility of selectively applying bounds checks to memory addresses that are either a) accessed by an external interface or user input, or b) written to (or both). This method, known as *taint-based bounds checking* has shown promise, effectively reducing the number of required bounds checks while still ensuring a high level of security. However, the original implementation proposed by Chuang *et al.* relied on the x86 `bounds` instruction. As a result, it was not portable to other architectures.

We present VAPOREON, a re-implementation of taint-based bounds checking as an LLVM pass. In doing so, we address the limitation of the original paper by extending the applicability of taint-based bounds checking to a wider range of architectures. This would allow the integration of this method into diverse range of systems, enhancing its scalability and effectiveness.

[1]HeartBleed Bug: https://heartbleed.com/

[2]National Vulnerability Database: [https://nvd.nist.gov/vuln/] (see CVE-2023-7024, CVE-2024-0519, CVE-2023-42916, etc.)

[3]As defined by Google in [1].

II. Related Work

Taint-based bounds checking is not a new concept; Chuang *et al.* originally proposed this method of bounds checking in 2007. Since then, the field of research focused on ensuring spatial safety in memory-unsafe languages has evolved significantly.

A large portion of this research builds on the primary concept introduced by Chuang *et al.*; that is, identifying and removing unnecessary bounds checks. Some examples include CHOP, which analyzes the profiling data of previous program executions to locate "safe zones" and remove unnecessary bounds checks [4]; the method introduced by Zhang *et al.*, which uses "super" data-flow graphs and an SMT solver to more precisely identify buffer overflow conditions that necessitate bounds checking [5]; and the tool developed by Situ *et al.*, which combines static analysis, taint analysis, and data-flow analysis to generate recommendations for enhancing code security [6]. While approaches like CHOP and Zhang *et al.* significantly reduce the amount of bounds checking in a program, they are fairly complex and incur a performance overhead of their own. Meanwhile, the tool developed by Situ *et al.* is efficient and covers a wider range of memory safety issues, but only offers recommendations for improving security—manual developer input is still required to make the program secure.

Another significant strand of research focuses on increasing overall efficiency—either in storing bounds information or handling invalid accesses (i.e. possible attacks). Techniques in this domain range from the compression of fat pointers (which may introduce inaccuracies) [7] to modifying the handling of function return addresses using the Last Branch Record (LBR) to thwart buffer overflow attacks [8]. Similar to the initial work by Chuang *et al.*, these methods often rely on specific architectural or hardware features.

Moreover, there have been efforts to automate the translation of C and C++ programs into memory-safe languages, such as Rust [9], [10]. While this approach provides convenience for smaller projects, the lack of full automation presents challenges for converting larger C and C++ codebases, rendering it impractical in many scenarios.

In summary, while the taint-based bounds checking is not a novel approach, it is still a relevant method that offers a simple and scalable run-time optimization on bounds checking. VAPOREON contributes an implementation of this method that is independent of specific architectural or hardware features.

III. Methods

Bounds checking verifies all pointer and array accesses. While this provides complete protection against all kinds of buffer overflow attacks, it has significant performance overhead. In this section, we present a technique to limit the scope of bounds checking to only those objects that are vulnerable to a buffer overflow attack. We focus on LLVM's capabilities in static analysis and program transformation to achieve efficient bounds-checking optimization.

A. Implementation Overview

VAPOREON streamlines bounds checking through a targeted approach that identifies vulnerable objects for optimization. It implements a multi-step process, starting with the detection of allocations at risk. The system then selectively applies bounds propagation to these identified objects. This method effectively minimizes performance overhead while maintaining security.

B. Allocation Identification

The identification and handling of `AllocaInst` instruction are crucial in our approach, as these LLVM instructions represent memory allocations on the stack, which are frequent targets for buffer overflow attacks. Our strategy involves two key phases: detecting `AllocaInst` instances and categorizing them based on their vulnerability to buffer overflows.

The initial step in processing `AllocaInst` instructions is their comprehensive detection throughout the function. We traverse each basic block within the function, examining each instruction to identify instances of `AllocaInst`. This methodical examination ensures that all stack allocations are captured for subsequent analysis.

Once an `AllocaInst` is identified, we categorize it based on the type of memory it allocates, which is crucial for determining the appropriate boundschecking strategy. Our first category is array allocations. These are particularly sensitive due to their common use in operations that can lead to buffer overflows. For each array-type allocation, we automatically determine its size by accessing the number of elements allocated, which LLVM stores as part of the `AllocaInst` itself. This size information is crucial for setting up accurate bounds. The second category is pointer allocations. For allocations intended to store pointers, further analysis is required to establish if they are used in a manner that might risk out-of-bounds access. These are tagged for additional processing during the bounds propagation phase.

For each identified `AllocateInst`, especially those involving array allocations, we establish initial bounds. This process involves creating metadata entries or additional instructions that record the base address (lower bound) and the extent (size) of the allocation. This metadata is crucial for subsequent bounds checking during program execution. Such a setup ensures that each stack allocation carries its bounds information throughout the function, allowing for effective and efficient runtime checks.

The detected and categorized `AllocateInst` instructions are then integrated into our LLVM optimization passes. By embedding bounds information directly within the LLVM intermediate representation, we enable other optimization passes to utilize and respect these bounds, thereby enhancing the overall security posture of the compiled program without compromising performance.

C. Bounds Propagation

Our implementation of bounds propagation is designed to maintain the integrity of bounds across all computations and transformations within a function. This process is facilitated through a breadth-first search (BFS) mechanism, which ensures comprehensive coverage and efficient propagation of bounds information. The propagation starts by initializing a queue with pointers that have known bounds, typically parameters and allocations that have been previously processed. Each pointer in the queue is examined for its uses throughout the function. For every use, we determine the type of instruction and apply the appropriate propagation logic.

When the use is a `PHINode`, we check if bounds for this node have already been established. If so, we update the existing bounds with the new information. If not, we initialize the `PHINode` with bounds derived from its incoming values. This ensures that the bounds are correctly merged from all incoming branches, preserving safety across control flow merges. for `StoreInst` and `LoadInst`, we handle the propagation of bounds to ensure that any stored or loaded values maintain their associated bounds. When a pointer is stored, the bounds are also stored alongside, allowing any subsequent loads to retrieve and apply the same bounds, thus maintaining integrity throughout the pointer's lifecycle within the function. For other instructions that manipulate pointers (i.e., `GetElementPtrInst`), we forward the bounds from the source pointer to the result of the instruction. This ensures that any computations derived from a bounded pointer also carry the bounds, preventing indirect out-of-bounds accesses.

During the propagation, we dynamically update the bounds information in the metadata attached to each instruction. This metadata tracks the lower and upper bounds for pointers, enabling runtime checks to be inserted accurately. The use of metadata ensures that the bounds information is tightly coupled with the pointer data, minimizing overhead and simplifying the management of bounds.

D. Bounds Checking

The first step in handling out-of-bounds operations is creating a basic block that contains a trap instruction. Since bounds checking will always require a conditional branch, it is more efficient to create one instance of the trap instruction rather than an arbitrary number.

With a trapping mechanism in place, it is now time to iterate over all basic blocks and instructions within our program, inserting code to bounds check whenever necessary. More specifically, we are only concerned with checking the bounds of store instructions whose operands have been added to our `bounds` map. This guarantees that we are able to look up the lower bound and size of the operands container, and gives us all the information necessary for generating the bound cheching code.

To facilitate the bounds checking, we create three instructions, a `sub`, an `icmp`, and a `br`. The subtraction instruction generates the difference between the store operand and the base pointer of the container. The comparison instruction then compares this difference to the size of the container, and branches according to which value is greater. Finally, the branch instruction branches to the trap basic block if the difference is larger than the size of the container, and branches to the store instruction otherwise. Since the store instruction must now be at the beginning of a basic block for us to branch to it, we must split the current basic block into two. The `splitBasicBlockBefore` function in llvm successfully does this, making our newly created `br` instruction the new terminator of the previous block. With these three new instructions and basic block, we now have all the code necessary for efficient bounds checking.

```
for each basic block:
  for each instruction:
    if store instruction:
      ptr = operand of store
      if ptr not in bounds map:
        continue
      lower, size = bounds[ptr]
      diff = ptr - lower
      flag = diff < size
      new_block = splitBasicBlockBefore(store)
      if flag:
        terminator = br new_block
      else:
        terminator = br trap_block
```

Listing 1: Pseudocode for bounds checking

IV. Evaluation and Analysis

A. Security

To evaluate the security improvements of VAPOREON, we ran a variety of vulnerable C programs that encapsulated various types of attacks on binary programs. First, we tested VAPOREON on Listing 2, a simple out-of-bounds access.

```
int main() {
    char buffer[4];
    buffer[6] = 'a';
}
```

Listing 2: Simple out-of-bounds access

```
daniel@tripledelete > ./example_outofbounds
*** stack smashing detected ***: terminated

daniel@tripledelete > ./example_outofbounds.vaporeon
terminated by signal SIGILL (Illegal instruction)
```

Since the machine tested had a wide range of protections turned on, including DEP, ASLR, and stack canaries, the original program was also able to catch this attack, albeit at a later stage. When compiled with VAPOREON, we see our own `trap` instruction being hit.

Next, we tested passing pointers as parameters. Since the C standard library was not recompiled with VAPOREON, we had to rewrite some functions that we would use, such as `strcpy`. Our next test was on Listing 3, where we call our own version of `strcpy`.

Again, we were able to observe our trap being hit, as VAPOREON passed fat pointers to `my_strcpy`, meaning that every access inside of that copy function was bounds checked.

```
#include <stdio.h>

void my_strcpy(char* dest, const char* src) {
    while ((*dest++ = *src++));
}

void vuln(char* source) {
    char buffer[8];
    my_strcpy(buffer, source);
}

int main() {
    char buffer[16];
    for (int i = 0; i < 15; ++i) {
        buffer[i] = 'A';
    }
    buffer[15] = '\0';
    vuln(buffer);
}
```

Listing 3: Program utilizing `strcpy` with buffers passed as parameters

```
daniel@tripledelete > ./example_parameter
*** stack smashing detected ***: terminated
daniel@tripledelete > ./example_parameter.vaporeon
terminated by signal SIGILL (Illegal instruction)
```

There are a wide variety of binary exploitation techniques, but those techniques often differ based on their methods used to bypass defenses trying to detect out-of-bounds writes after they have happened. Since we prevent any out-of-bounds writes from occuring, we did not feel the need to test all of these vulnerabilities.

However, there is one nefarious attack that no "post-mortem" defense can prevent: a data-based attack. By overwriting a buffer on the stack, we can overwrite the values of variables after them. Unless a canary is put in place after every buffer (which may make the canary more prone to being leaked and add additional overhead at the end of functions), we will have no way of detecting this. An example of this attack is shown in Listing 4.

```
#include <stdio.h>

int main(int argc, char** argv) {
    char buf[16];
    char buf2[16];
    buf[0] = 1;
    buf2[0] = 3;
    if (argc != 1) buf2[16] = 8;
    if (buf[0] != 1) {
        puts("this is a restricted area!");
    } else {
        puts("normal operation");
    }
}
```

Listing 4: Data-based attack

As shown in the code, we have two buffers on the stack. `buf` contains a "secret" value `1`, while `buf2` is manipulable by the user. If a parameter is passed in, we index out of bounds of `buf2`, overwriting the first element of `buf` to be `8`. This leads to the `if` check succeeding.

```
daniel@tripledelete > ./example_data_attack
normal operation
daniel@tripledelete > ./example_data_attack vuln
this is a restricted area!
```

Note that even on a machine with full defenses against binary exploitation, there is no "stack smashing detected" message that we saw previously. This attack goes completely unnoticed, since we do not try to jump to the stack, or write past the end of the stack frame at all. However, if we run it through VAPOREON, we see that the fat pointer catches out-of-bounds accesses:

```
daniel@tripledelete > ./example_data_attack.vaporeon
normal operation
daniel@tripledelete > ./example_data_attack.vaporeon
vuln
terminated by signal SIGILL (Illegal instruction)
```

B. Performance

To measure the performance impact of VAPOREON, we measured two key aspects of bounds checking: bounds checking writes, and propagating bounds checks. First, we measure the overhead of writes.

```
int main() {
    char buffer[1024];

    for (int i = 0; i < 1000000000; ++i) {
        buffer[i % 1024] = 'a';
    }
}
```

Listing 5: Testbench for write overhead

In Listing 5, we initialize a buffer, and then write to it repeatedly across the entire array. Running this on a Intel Xeon E-2334 processor gave an 8.7% overhead to bounds checking. This slowdown is much lower than what was observed for the one-branch approach, we believe that this is due to the single fat pointer leading to minimal read-write traffic during bounds checking (since we keep these bounds in registers as often as possible), and good branch prediction leading to high throughput of this loop.

Next, we measured the effect of propagating bounds checks. Since we ran our programs at `-O0`, we had many stores and loads of local variables. Our code is shown in Listing 6.

```
#include <stdio.h>

void print_first_char(char* p) {
    char c = p[0];
    putchar(c);
}

int main() {
    char buffer[16];
    buffer[0] = '!';
    char* a;
    char* b;
    char* c;
    a = buffer;
    for (int i = 0; i < 100000000; ++i) {
        b = a;
        c = b;
        a = c;
    }
    a[16] = 'a';
    print_first_char(a);
}
```

Listing 6: Testbench for propagation overhead

Here, we observe a significant slowdown at runtime: roughly a 2x runtime. We attribute this slowdown to the constant loads and stores from memory that `-O0` compiles into our code: at each store, we must also store the related bounds information to memory, adding two more stores and loads. If these variables were stored in memory, we would likely have minimal overhead, since we propagate their bounds at compile time, meaning that the last access to `a` would utilize the bounds calculated for `buffer`. However, we note that this slowdown cannot be avoided for function calls, where we need to re-package the fat pointer to be passed in. This is consistent with benchmarks such as `parser`, where Chuang *et al.* observed nearly a 100% slowdown on the benchmark.

V. Conclusion

VAPOREON contributes to the advancement of memory safety for C/C++ programs through its targeted and optimized bounds checking. By selectively applying bounds checking to stack objects deemed vulnerable to buffer overflow attacks, it effectively reduces the performance overhead typically associated with such security measures. This LLVM-based implementation not only enhances security but also ensures broad applicability across various architectures, making it a scalable solution for contemporary computing environments. Moreoever, its ability to prevent even sophisticated binary exploitation attacks prior to their execution solidifies its value as a crucial tool in the arsenal against memory corruption vulnerabilities. Overall, VAPOREON successfully balances the trade-offs between security and performance, demonstrating its potential as an essential enhancement in systems where robustness and efficiency are paramount.

References

[1] A. Rebert and C. Kern, “Secure by Design: Google's Perspective on Memory Safety,” 2024.

[2] A. One, “Smashing the stack for fun and profit,” *Phrack magazine*, vol. 7, no. 49, pp. 14–16, 1996.

[3] W. Chuang, S. Narayanasamy, B. Calder, and R. Jhala, “Bounds checking with taint-based analysis,” in *Proceedings of the 2nd International Conference on High Performance Embedded Architectures and Compilers*, in HiPEAC'07. Ghent, Belgium: Springer-Verlag, 2007, pp. 71–86.

[4] Y. Chen, H. Xue, T. Lan, and G. Venkataramani, “CHOP: Bypassing runtime bounds checking through convex hull Optimization,” *Computers & Security*, vol. 90, p. 101708–101709, 2020, doi: https://doi.org/10.1016/j.cose.2019.101708.

[5] Y. Zhang, L. Chen, X. Nie, and G. Shi, “An Effective Buffer Overflow Detection With Super Data-Flow Graphs,” in *2022 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, 2022, pp. 684–691. doi: 10.1109/ISPA-BDCloud-SocialCom-SustainCom57177.2022.00093.

[6] L.-Y. Situ, L.-Z. Wang, Y. Liu, B. Mao, and X.-D. Li, “Automatic Detection and Repair Recommendation for Missing Checks,” *Journal of Computer Science and Technology*, vol. 34, no. 5, pp. 972–992, Sep. 2019, doi: 10.1007/s11390-019-1955-3.

[7] S. Xu, E. Liu, W. Huang, and D. Lie, “MIFP: Selective Fat-Pointer Bounds Compression for Accurate Bounds Checking,” in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, in RAID '23. <conf-loc>, <city>Hong Kong</city>, <country>China</country>, </conf-loc>: Association for Computing Machinery, 2023, pp. 609–622. doi: 10.1145/3607199.3607212.

[8] H. Zhou, K. Kang, and J. Yuan, “HardStack: Prevent Stack Buffer Overflow Attack with LBR,” in *2019 International Conference on Intelligent Computing, Automation and Systems (ICICAS)*, 2019, pp. 888–892. doi: 10.1109/ICICAS48597.2019.00191.

[9] P. Larsen, “Migrating C to Rust for Memory Safety,” *IEEE Security & Privacy*, vol. 0, no. , pp. 2–9, 2024, doi: 10.1109/MSEC.2024.3385357.

[10] A. E. Michael *et al.*, “MSWasm: Soundly Enforcing Memory-Safe Execution of Unsafe Code,” *Proc. ACM Program. Lang.*, vol. 7, no. POPL, Jan. 2023, doi: 10.1145/3571208.