# Memory safety in C by abstract interpretation

Joseph Jones
Brigham Young University
Provo, UT 84602
josephjjones@byu.edu

James Wasson
Brigham Young University
Provo, UT 84602
jamesjameswasson@byu.edu

Seth Poulsen
Brigham Young University
Provo, UT 84602
poulsenseth@yahoo.com

Sean Brown
Brigham Young University
Provo, UT 84602
seangb@byu.edu

Peter Aldous
Brigham Young University
Provo, UT 84602
aldous@cs.byu.edu

Eric Mercer
Brigham Young University
Provo, UT 84602
egm@cs.byu.edu

## ABSTRACT

Pointer arithmetic is a core feature of the C programming language and C program analysis is impossible without an understanding of its effects. Many program analyses opt to be unsound in the presence of pointer arithmetic or preserve soundness at the cost of precision. However, the number of operations that can be performed safely on pointers is actually quite small. As was observed by Might et al. [11], these few operations can be precisely modeled with a simplified Peano arithmetic. This paper presents an interpreter that uses a memory model based on this arithmetic. It desugars C programs to a simple imperative language using standard semantics-preserving techniques to simplify the interpretation. The result is a prototype analysis that reasons precisely about memory safety in full C programs without programmer annotations.

## Keywords

Abstract interpretation, pointer analysis

## 1. INTRODUCTION

Pointer arithmetic is an essential part of the C programming language but is a hobgoblin of program analyses. By manipulating pointers, programmers have fine-grained control over their software. However, errors in pointer arithmetic can be both catastrophic and difficult to find. Pointer arithmetic errors are especially difficult to find in a static analysis, where conventional abstractions for reasoning about values do not naturally capture pointer behavior. Several of these analyses are described in Section 5.

Although pointers may be used in many ways, the number of safe pointer operations that do not rely on a particular implementation of the C language is quite small. Many pointer operations rely on detailed information external to the C specification; in a classic turn of phrase, this is "unwarranted chumminess with the compiler" [10].

Motivated by this observation, Might et al. [11] present a model of memory based on a simplified Peano arithmetic that is suitable for reasoning about C pointers. This arithmetic uses successors as in Peano arithmetic but defines only addition and subtraction. By defining successors and predecessors so that separately allocated regions of memory are unrelated, the memory model permits only safe pointer operations.

This work presents YAAM, an interpreter that implements the memory model of Might et al. [11]. It uses the CESK [6] semantic

```
#include <stdio.h>

void f(int *x) {
    printf("0x%x\n", *(x+3));
}

int main(void) {
    long s[2];
    s[0] = 0xEFBEADDE00000000;
    s[1] = 0xFFFFFFFF03020100;
    int *p = (int *)&s;
    f(p + 0); // safe
    f(p + 1); // unsafe
    return 0;
}
```

**Figure 1: Safe and unsafe pointer arithmetic on an array**

model, allowing it to be systematically transformed into an abstract interpreter [13, 9]. YAAM is capable of executing programs as defined by the C99 standard [1]. YAAM can reason soundly and precisely about pointer arithmetic and can identify unsafe memory accesses. Although YAAM does perform a concrete interpretation, it was built with the intention to later perform a static analysis through abstraction.

The contributions of this work are:

- A C interpreter based on the memory model of Might et al. [11] that can be lifted to an AAM-style abstract interpreter.
- A demonstration that this memory model permits precise and permissive reasoning about pointer arithmetic.

The structure of this paper is as follows: Section 2 elaborates on the problem to be solved with specific examples. Then, Section 3 gives a brief formal presentation of the analysis. Section 4 discusses preliminary empirical results in the form of a case study. Section 5 follows with a summary of relevant related work. Finally, Section 6 concludes.

## 2. EXAMPLE

Figure 1 and Figure 2 demonstrate pointer arithmetic in two forms. The program in Figure 1 manipulates bytes in an array

```
1  #include <stdio.h>
2
3  struct A{
4      int x;
5      int y;
6      int z;
7  };
8
9  struct B{
10     int x;
11     int y;
12 };
13
14 void f(void* a, int offset){
15     printf("0x%x\n", *((int*)a + offset));
16 }
17
18 int main(){
19     struct A first = {1,2,3};
20     struct A second = {4,5,6};
21     struct B third = {7,8};
22     int* a = (int*)&first;
23     int* b = (int*)&(first.z);
24     int offset = b - a;
25     f(&second, offset); // safe
26     f(&third, offset); // unsafe
27     return 0;
28 }
```

**Figure 2: Safe and unsafe pointer arithmetic on structs with different sizes**

and the program in Figure 2 calculates offsets to fields in structs. Both programs allocate memory in one function, add an offset of some sort, and read from memory. Each program demonstrates both a safe memory access and an unsafe memory access. These examples assume longs and ints are 8 and 4 bytes respectively.

Differentiating between safe and unsafe reads of this variety can be difficult to accomplish programmatically. Naïve type systems are ill-equipped to reason about reinterpretations of data. Additionally, classical static analyses are intraprocedural; as a result, they tend to be imprecise with respect to aliasing.

Abstract interpretation can address both of these limitations. An interprocedural abstract interpretation can reason soundly about aliasing [5, 8]. Moreover, an abstract domain based on simplified Peano arithmetic can address byte reinterpretation between integers and pointers. While its precision is necessarily limited, it can be sufficiently precise to yield useful results.

The remainder of this section describes the memory model intuitively. Section 3 presents it formally.

Ordinary stack variables have no specified successors or predecessors; their location with respect to each other is not specified. However, regions of memory allocated (e.g., with `malloc`) are guaranteed to be contiguous. Accordingly, allocation causes the successor and predecessor maps to update the address for each byte allocated so that it is related to its neighbors. Addresses on either end are related to special addresses for overflow and underflow.

In Figure 1, line 8, YAAM allocates memory for an array with two longs as a single region of size 16. YAAM stores the pointer `p` as a pointer to the beginning of `s` and can now read `s` one byte at a time, beginning at `p`. It does so using the successor map. An 8-byte read on `p` with an offset of 8 bytes (line 12) will be safe because 16 successors are defined for `s`. However, beginning an 8-byte read at an offset of 12 bytes (line 13) will move the pointer beyond the allocated region.

$$\begin{aligned}
prgm \in Prgm &:= [vdecl \mid fdecl \mid sdecl \mid td]^* \\
td \in TD &:= \mathsf{typedef}\ type\ id; \\
sdecl \in Sdecl &:= \mathsf{struct}\ id\ \{\ [type\ id;]^*\ \}; \\
vdecl \in Vdecl &:= id\ =\ rvalue; \\
decl \in Decl &:= type\ assign \\
assign \in Assign &:= lvalue\ =\ rvalue; \\
lvalue \in Lvalue &:= [\mathsf{*}]?id \\
rvalue \in Rvalue &:= [cast]?\ [ae \mid bexp \mid uexp] \\
cast \in Cast &:= (type) \\
ae \in AE &= ID + Literal \\
bexp \in BExp &:= ae\ binop\ ae \\
uexp \in UExp &:= unop\ ae \\
fdecl \in Fdecl &:= type\ id(\ [type\ id\,[,\ type\ id]^*]?)\ \{\ [stmt]^*\ \} \\
stmt \in Stmt &:= [label{:}]?\ lvalue\ =\ rvalue; \\
&\quad \mid [label{:}]?\ \mathsf{if}\ (ae)\ stmt\ [\mathsf{else}\ stmt]? \\
&\quad \mid [label{:}]?\ \mathsf{goto}\ label; \\
&\quad \mid [label{:}]?\ \ [lvalue\ =]?id(\ [ae\,[,\ ae]^*]?); \\
&\quad \mid \{\ [stmt]^*\ \}
\end{aligned}$$

**Figure 3: Surface syntax for desugared C language**

For Figure 2, YAAM applies the same offset (line 24) to two pointers to structs A and B, which have different sizes, by using the same method. YAAM safely reads struct `second` on line 25 because the pointer passed to `f` is incremented to `second.z` (line 6) and `second` is sufficiently large. YAAM finds the unsafe read (line 26) in struct `third` because the pointer to `third` is offset beyond the end of the memory allocated for `third`.

## 3. CONCRETE INTERPRETATION

Instead of interpreting C directly, YAAM desugars C programs to the language given in Figure 3. This language lacks complex expressions, arrays, struct references, etc. Instead, it uses simple expressions with temporary variables and byte-addressed pointer arithmetic. All casts are explicit. The semantics of this language are easily understood and formalized. However, since the desugared code is still C, it is easy to preserve certain high-level language constructs if their semantics help simplify a given analysis. For example, it can keep OpenMP constructs intact to help simplify analyses of concurrent programs.

YAAM's semantics are based on the CESK machine [6], a powerful general-purpose semantic model that represents explicit states as tuples of control (C), environment (E), store (S), and continuation (K). As a result, they can be systematically transformed into an abstract interpreter [13] and optimized [9]. The CESK state space used in YAAM is given in Figure 4.

### 3.1 Memory model

$$\begin{aligned}
\varsigma \in \Sigma &= \mathit{Stmt} \times E \times K \\
e \in E &= \mathit{Var} \rightharpoonup A \\
\sigma \in S &= A \rightharpoonup \mathit{Val} \\
\kappa \in K &= \mathit{Stmt} \times K \\
\mathit{val} \in \mathit{Val} &= A + \mathbb{Z} \\
a \in A &= \mathbb{Z}^{+} \\
l \in L &= \{\mathsf{byte},\ \mathsf{short},\ \mathsf{int},\ \mathsf{long}\} \\
\mathit{var} \in \mathit{Var} &= \text{the set of variables in the program} \\
\mathit{stmt} \in \mathit{Stmt} &= \text{the set of statements in the program}
\end{aligned}$$

**Figure 4: CESK state space**

This section summarizes the memory model of Might et al. [11]. Memory is allocated as some number of contiguous bytes. The C specification does not specify where the region lies in memory with respect to other memory regions [1]. For example, the execution of `malloc(n)` returns a pointer to the beginning of a region of $n$ contiguous bytes. No information is available regarding what comes before this pointer or more than $n$ bytes after this pointer.

Allocation is defined with a successor map $\sigma_+$ and a predecessor map $\sigma_-$. To allocate $n$ bytes, fresh addresses $a_1, a_2, \ldots, a_n$ are each pointed to a byte in the newly allocated space. $\sigma_+$ is updated so that $\sigma_+(a_1) = a_2$ and so on. Similarly, $\sigma_-(a_2) = a_1$. Formally,

$$\begin{aligned}
\sigma'_+ &= \sigma_+\left[a_1 \mapsto a_2, \ldots, a_{n-1} \mapsto a_n, a_n \mapsto a_\top\right] \\
\sigma'_- &= \sigma_-\left[a_n \mapsto a_{n-1}, \ldots, a_2 \mapsto a_1, a_1 \mapsto a_\perp\right].
\end{aligned}$$

The overflow address $a_\top$, the underflow address $a_\perp$, the null address *null*, and the undefined pointer $\top$ are all unsafe; reading from or writing to these pointers leads to an error state. Universally, $\sigma_+$ and $\sigma_-$ map unsafe pointers to $\top$.

For convenience, the predicate *safe* is defined over pointers:

$$\mathit{safe}(p) \equiv p \notin \{a_\top, a_\perp, \mathit{null}, \top\}.$$

## 3.2 Peano arithmetic for pointers

Given $\sigma_+$ and $\sigma_-$ for addresses, it is possible to compute the result of valid pointer arithmetic using a simplified Peano arithmetic that contains addition and subtraction but not multiplication. Operations such as multiplication and division always result in an unsafe pointer. Addition and subtraction of an integral value to a pointer are performed by calling the predecessor or successor map repeatedly. The difference of two pointers results in a integer offset that is equal to the number of steps in the $\sigma_+$ or $\sigma_-$ map it takes to reach the other. Because all pointers are desugared to byte pointers, no type information is required to add to them. The arithmetic is shown in Figure 5, where p is a pointer, n is an integer, and $\sigma_+^*$ is the transitive closure of $\sigma_+$.

Because Peano arithmetic was used in conjuction with the successor and predecessor maps, disjoint regions are unreachable by pointer arithmetic. This separation allows for the detection of unsafe pointer arithmetic that might rely on a compiler specific allocation scheme. In many cases, this permits precise reasoning even after arithmetic or aliasing has obfuscated the pointer.

$$\begin{aligned}
p + n \equiv n + p &\equiv \begin{cases} \sigma_+(p) + (n-1) & n > 0 \\ p & n == 0 \\ \sigma_-(p) + (n+1) & n < 0 \end{cases} \\
p - n &\equiv p + (-n) \\
p_1 - p_2 &\equiv \begin{cases} 0 & p_1 == p_2 \\ (\sigma_+(p_1) - p_2) + 1 & p_1 \sigma_+^* p_2 \\ (\sigma_-(p_1) - p_2) - 1 & p_2 \sigma_+^* p_1 \\ \top & \text{otherwise} \end{cases} \\
p \pm p &\equiv n - p \equiv \top
\end{aligned}$$

**Figure 5: Peano Pointer Arithmetic**

However, there are limitations to its precision; the shared overflow address cannot have a safe predecessor. In a concrete system, it is possible to reason precisely about such a reversal, but abstraction makes this impossible to do generally. Similarly, it is possible to coerce a pointer to an integral value and perform a series of multiplications, divisions, etc. that transforms it to something safe. However, this is, at best, a software development practice that is not encouraged.

## 3.3 Concrete semantics

With the memory model, it is possible to articulate the semantics of the interpreter. Pointer safety is formalized with a predicate *safe*, which evaluates whether or not a pointer is safe based on the definition given in Section 3.2. A region of memory beginning at $p$ that contains $n$ bytes is safe if the pointer to each byte in the region is safe. Formally, this is expressed with $\mathit{safe}_r$:

$$\mathit{safe}_r(p, n) \equiv n = 0 \vee (\mathit{safe}(p) \wedge \mathit{safe}_r(p+1, n-1)).$$

The transition relation $\rightarrow \subseteq \Sigma \times \Sigma$ relates states with their successors. The state space is formalized in Figure 4. As an example, the transition rules for assignment, specialized for assignment from a pointer to a variable, are included here. The transition relation makes use of $n$, which returns the syntactic successor to the given statement:

$$\frac{\sigma' = \sigma\left[e(id_d) \mapsto \sigma(p)\right] \quad p = e(id_p) \qquad \mathit{stmt} = id_d = {*}id_p \quad \mathit{safe}_r(p, \mathit{sizeof}(id_d))}{(\mathit{stmt}, e, \sigma, \kappa) \rightarrow (n(\mathit{stmt}), e, \sigma', \kappa)}$$

$$\frac{\mathit{stmt} = id_d = {*}id_p \quad \neg\mathit{safe}_r(p, \mathit{sizeof}(id_d)) \quad p = e(id_p)}{(\mathit{stmt}, e, \sigma, \kappa) \rightarrow \mathit{error}}$$

## 3.4 Abstracting the memory model

In order to build an abstract interpreter from this concrete interpreter, the CESK state space can be lifted to an abstract state space. This means that the addressing scheme must be changed so that addresses come from a finite set and the successor function updated accordingly. This can be accomplished using any abstraction function $\alpha : \mathbb{Z}^+ \rightarrow \hat{\mathbb{Z}}^+$ such that $\hat{\mathbb{Z}}^+$ is finite. The abstract successor function $\hat{\sigma}_+$ is induced from $\alpha$ and $\sigma_+$ (as is the abstract predecessor function $\hat{\sigma}_-$ from $\alpha$ and $\sigma_-$). One possible definition for $\alpha$ is modular arithmetic: $\alpha(n) = n \pmod{m}$. As with any abstract domain, it is defined on a join-semilattice. In the case of modular arithmetic, the join of any two distinct values is $\top$.

Regardless of the definition of $\alpha$, memory allocation is as expected. When $n$ bytes are allocated, a series of $n$ adjacent addresses is selected from $\hat{\mathbb{Z}}^{+}$. The stores are updated (weakly, so that existing values are joined with new values) as follows:

$$\hat{\sigma}'_{+} = \hat{\sigma}_{+} \sqcup [\alpha(a_1) \mapsto \alpha(a_2), \ldots, \alpha(a_{n-1}) \mapsto \alpha(a_n), \alpha(a_n) \mapsto a_{\top}]$$
$$\hat{\sigma}'_{-} = \hat{\sigma}_{-} \sqcup [\alpha(a_n) \mapsto \alpha(a_{n-1}), \ldots, \alpha(a_2) \mapsto \alpha(a_1), \alpha(a_1) \mapsto a_{\bot}]$$

Because all other aspects of the CESK space are standard, they become parts of an abstract interpreter using the system presented by Van Horn and Might [13].

## 4. EMPIRICAL STUDY

A concrete interpreter, YAAM, was used to verify the memory model with a small set of benchmarks. YAAM correctly identifies the memory access errors for the programs provided in Figure 1 and Figure 2, programs with memory access errors are labeled as unsafe in Table 1.

### 4.1 Methodology

To implement YAAM the labor was divided into three main parts: parsing, desugaring, and interpretation. Pycparser, an open source c parser in python, was used to parse the c files into an abstract syntax tree (AST), which can be used flexibly.

Then, a series of AST traversals desugar the AST to the grammar shown in Figure 3. Instead of desugaring all in one go, small changes are made one at a time such as for and do while loops are transformed to while loops then while loops are transformed to goto statements. Array and struct references are desugared to pointer arithmetic, then the pointer arithmetic is reduced to byte pointer arithmetic.

After the desugaring is complete, the interpreter traverses the AST to find main and set up global variables, types, and functions. Following the CESK model, YAAM creates a single state and evaluates each state until the halt state is invoked. Since YAAM is concrete, updates to the store are strong, and a single next state is generated for each state evaluated; this is abstractable by allowing for multiple next states to evaluated and bounding the address space for frame, heap, and continuation addresses. If an error occurs then interpretation is interrupted, and the error is reported.

The case study was executed remotely with an Intel(R) Xeon(R) Gold 5120 processor running at 2.20GHz using 7.7 GB of RAM, running Ubuntu 18.04.1 LTS.

### 4.2 Results

Table 1 contains timing data in seconds for some simple programs. The program in Figure 1 is called Array cast and the program in Figure 2 is called Struct offset. The MxN Array programs simply iterate over a multidimensional array twice, once to assign then once more to print.

### 4.3 Analysis

The study shows that parsing and desugaring is relatively quick, while interpreting dominates the execution time. As the length of interpretation increases, the time increase is super-linear because of the large number of states in memory.

## 5. RELATED WORK

This tool is based on a simplified Peano arithmetic model originally developed by Might et al. [11]. It uses the CESK [6] machine as a semantic model, which can be systematically transformed into an abstract interpreter [13, 9].

Balatsouras and Smaragdakis [2] present a points-to analysis for C programs that is sound for many use cases. In their analysis, not all language features are addressed. One such language feature that they do not address is casting.

Cohen et al. [3] present an analysis of C programs that uses SMT solvers to reason about the memory model. It is general and powerful but does not preserve boundaries between regions of memory that are allocated separately, potentially missing errors that depend on memory allocation behavior. The SeaHorn [7] framework uses SMT solvers to reason about LLVM bitcode transformed into Horn clauses.

Conway et al. [4] reason about soundness on the assumption that the program is memory-safe and show that several other analyses are sound under this assumption. The memory safety analysis they propose operates on a C-like language that lacks control flow constructs of any variety.

Ströder et al. [12] use symbolic execution to prove memory safety in LLVM bitcode. They ignore integer overflows and constrain variable instantiation. Their analysis answers questions about memory safety, as does this analysis. However, their analysis does as many do in that it allows the layout of allocated memory to affect the detection of unsafe memory accesses. As such, it can be overly permissive.

## 6. CONCLUSION

This analyses demonstrably permits precise and permissive reasoning about pointer arithmetic. YAAM, being based on the memory model of Might et all. [11], interprets arbitrary C programs, reasons soundly and precisely about pointer arithmetic, and can identify unsafe memory accesses. Future work includes abstraction and other transformations of the CESK model presented for the purpose of interpretation.

The empirical study shows the simplified Peano arithmetic of Might et al. [11] allows for byte level operations to be performed precisely. This memory model allows for verification of memory safety. Additionally, it shows that strict aliasing of pointers is not required, allowing pointers of different types to have the same address. This proof of concept demonstrates that such an analysis is viable on C programs, even when they use pointer arithmetic.

Being a CESK interpreter, YAAM can be systematically transformed through methods presented by Van Horn and Might [13] to an abstract interpreter. Such an abstract interpreter could not only reason about memory accesses in C but could also be used as a general purpose analyzer.

YAAM can be used as the basis for an explicit state model checker. Its search space could include inputs or, in the presence of concurrency, schedules. It is also suitable for lightweight analyses, such as classical k-CFA, for use on larger code bases. For example, ongoing work includes implementation of k-CFA for verification of security properties in OpenSSL.

## 7. ACKNOWLEDGMENTS

| Application | Parsing (s) | Desugaring (s) | Interpretation (s) | Total time (s) |
|---|---|---|---|---|
| Array cast (unsafe) | 0.798 | 0.062 | 0.008 | 0.870 |
| Array cast | 0.789 | 0.066 | 0.010 | 0.865 |
| Struct offset (unsafe) | 0.784 | 0.119 | 0.015 | 0.919 |
| Struct offset | 0.787 | 0.115 | 0.016 | 0.920 |
| 10x10 Array | 0.947 | 1.003 | 0.371 | 2.322 |
| 10x100 Array | 0.894 | 0.977 | 5.289 | 7.161 |
| 100x100 Array | 0.921 | 1.059 | 248.804 | 250.785 |
| 100x1000 Array | 0.935 | 0.989 | 20542.441 | 20544.366 |

**Table 1: Empirical results from the case study**

This work is supported by the National Science Foundation under grant 1302524.

## 8. REFERENCES

[1] Final version of the c99 standard with corrigenda tc1, tc2, and tc3 included, formatted as a draft. http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf, 2007.

[2] G. Balatsouras and Y. Smaragdakis. Structure-sensitive points-to analysis for c and c++. In *International Static Analysis Symposium*, pages 84–104. Springer, 2016.

[3] E. Cohen, M. Moskal, S. Tobies, and W. Schulte. A precise yet efficient memory model for c. *Electronic Notes in Theoretical Computer Science*, 254:85–103, 2009.

[4] C. L. Conway, D. Dams, K. S. Namjoshi, and C. Barrett. Pointer analysis, conditional soundness, and proving the absence of errors. In M. Alpuente and G. Vidal, editors, *Static Analysis*, pages 62–77, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[5] A. Deutsch. Semantic models and abstract interpretation techniques for inductive data structures and pointers. In *Proceedings of the 1995 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 226–229. ACM, 1995.

[6] M. Felleisen and D. P. Friedman. A reduction semantics for imperative higher-order languages. In *Proceedings of the Parallel Architectures and Languages Europe, Volume I*, pages 206–223, London, UK, UK, 1987. Springer-Verlag.

[7] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The seahorn verification framework. In D. Kroening and C. S. Păsăreanu, editors, *Computer Aided Verification*, pages 343–361, Cham, 2015. Springer International Publishing.

[8] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61. ACM, 2001.

[9] J. I. Johnson, N. Labich, M. Might, and D. Van Horn. Optimizing abstract abstract machines. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 443–454, New York, NY, USA, 2013. ACM.

[10] B. W. Kernighan and D. Ritchie. *The C Programming Language*. Prentice Hall Software Series, second edition, 1988.

[11] M. Might, O. Shivers, B. Chambers, and S. Strickland. Abstract interpretation of imperative programs using garbage-collectable pointer arithmetic. Private communication, 2007.

[12] T. Ströder, J. Giesl, M. Brockschmidt, F. Frohn, C. Fuhs, J. Hensel, P. Schneider-Kamp, and C. Aschermann. Automatically proving termination and memory safety for programs with pointer arithmetic. *Journal of Automated Reasoning*, 58(1):33–65, Jan 2017.

[13] D. Van Horn and M. Might. Abstracting abstract machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 51–62, New York, NY, USA, 2010. ACM.