NATIONAL UNIVERSITY OF SINGAPORE

ASSIGNMENT II REPORT - CS5231

ANALYSIS OF MEMORY SAFETY TOOLS IN C

Group 5

Vijayanand Thangavelu (A0169371X)
Sonal Devadas Shenoy (A0174450J)
Vipul Sharma (A0178385M)

School of Computing

October 25, 2018

Analysis of Memory Safety Tools in C

https://github.com/vijathanga/memory-safety

Vijayanand Thangavelu Vipul Sharma Sonal Devadas Shenoy

I. Introduction

C is the building block of many high-level languages and softwares like operating systems, browsers and device drivers. Due to its high performance and explicit memory management, C is still one of the most widely used languages. But this high performance comes at the expense of memory safety, which makes C a memory unsafe language. At a high level, memory safety is ensuring safe and valid access of memory location when performing memory related operations. Failing to do so causes memory errors, which can lead to undefined program behaviour. These memory errors can be broadly classified into spatial and temporal memory errors. Spatial memory error occurs when memory access happens outside the permitted bounds of target buffer - like accessing out of bound index of an array. Buffer overflow and over-read, invalid array index, invalid pointer are few spatial memory violations. Temporal memory violation happens when a deallocated memory is accessed using a pointer. This includes dangling pointer, illegal free, wild pointers, double free and null pointer dereference. These memory errors are typical in C and are laborious to debug.

To detect these memory errors in C, various tools and techniques have been developed over the years. In this paper, two such memory safety tools for C - Splint and SoftBound+CETS are analyzed and evaluated.

II. Design and Techniques

A. Splint

Splint, short for "Secure Programming Lint", is a lightweight static analysis tool for detecting anomalies and security violations in C code [1]. It does flow-sensitive analysis at the procedural level, instead of doing it for the entire code. This makes Splint to analyze code faster, as flow analysis for entire code is complex and time-consuming. Splint does analysis by extracting information from annotations specified by programmer in the code. These annotations are nothing but stylized C comments, that describe assumptions about function state, parameters, global variables, return variables, variable's lifetime etc. Since these annotations are enclosed in comments, the compiler ignores them during compilation. For instance, `/*@notnull@*/` annotation before a function parameter indicates that parameter value can not be null at call location of the function, else a warning is generated. This kind of state information gives scope for splint to detect various kinds of vulnerabilities and enforce checks in code.

Spatial Errors: Splint considers buffer as contiguous memory and assigns two attributes `maxSet` and `maxRead` [2]. `maxSet` of a buffer is its highest index that can be used as lvalue (memory where valid write can be done). `maxRead` of a buffer is its highest index that can be used as rvalue. In other words, `maxRead` denotes the highest index of a buffer that has initialized data. Value of `maxRead` is always equal to or less than `maxSet`.

For instance, for buffer `int a[MAX]`, at the time of declaration `maxSet(a) = MAX-1` and `maxRead(a) = 0`. When a buffer is accessed as lvalue or rvalue, Splint generates constraints based on `maxSet` and `maxRead` attributes. For the statement `a[j] = x`, the buffer must meet the precondition constraint `maxSet(a) >= j`. If Splint is unable to resolve pre-constraint or if it's not met, a warning is issued to denote possible buffer overflow error. Similar to `maxSet` and `maxRead`, there are `minSet` and `minRead` that denotes lowest buffer index that can be used as lvalue and rvalue respectively.

For functions, there are two clauses namely `requires` and `ensures`. Conditions in `requires` clause should be satisfied before the function call. Conditions in `ensures` clause denote the state of variables after the function returns.

```
char* strcpy (char *a, const char *b)
 /*@requires maxSet(a) >= maxRead(b)@*/
 /*@ensures maxRead(a) == maxRead(b)@*/
```

Consider the above library function `strcpy` from [3]. The `required` clause checks if buffer `a` has enough memory to hold initialized data in buffer `b`. Warning is issued if `a` doesn't have enough memory or if Splint cannot resolve pre-contraint. After function execution, `b` is copied to `a`, hence condition `maxRead(a)` == `maxRead(b)` holds good. This is denoted by the `ensure` clause. These constraints ensures that out of bound access of buffer are exposed by Splint.

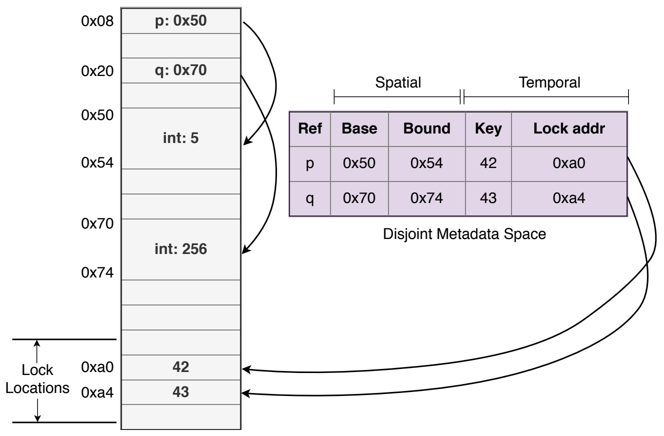
Null pointer dereferencing is a frequent cause of program failures. Splint identifies this problem by checking if a pointer is null wherever the pointer is dereferenced by using information obtained from annotations. For instance, if a pointer is annotated as `/*null*/` it indicates that the pointer can have null values. Splint looks for null check before dereferencing this pointer, else a warning is generated.

Temporal Errors: If an undefined memory is used as rvalue, it can cause undefined program behaviour. Splint raises warnings for such errors. It also does type checking and issue warnings if an `int` type is type-casted to `char`. This enables Splint to detect integer overflows and signed integer errors. It also issues warning if reference is made to a stack variable, that has scope outside variable's function. This is known as *stack reference* and it can cause undefined behaviour of program.

When memory is deallocated without proper checks, it can lead to deallocation errors - (1) deallocating memory when there are live pointers still referencing that memory (dangling pointers) (2) failing to free memory before last reference to it is lost (memory leak). In Splint, these errors are detected using a concept *obligation to release storage*. Each reference has an obligation to release its resource. This obligation can be transferred through function returns, parameters or external reference assignment. There are many types of reference qualifiers. `only` annotation is used when the reference is the only pointer to the object it points to and it has the obligation to release storage. Else, they can transfer the obligation through one of the above mentioned methods. `temp` annotation is used for function parameters that is only used temporarily by the function and they don't have obligation to release storage. Whenever a `temp` reference releases storage or is aliased, error is raised. `owned` annotation, similar to `only` has the obligation to release storage, but is not the only reference pointing to the memory. There can be other `dependent` references pointing to same location, however these `dependent` references don't have the obligation to release storage. `shared` reference can be used in occasions where there are multiple references pointing to an object which is never released during the program lifetime. Splint also has support for enforcing checks on reference counting objects. These checks ensure that reference count of these objects are incremented/decremented whenever reference to them is created/destroyed. All these checks helps to ensure temporal memory safety in C.

Splint also has support user defined annotations. One such useful annotation is the taintedness attribute of a `char` reference. `char` reference from untrusted sources like user input is marked as tainted. If such tainted reference is used in place where untainted reference is expected, Splint raises an error. Any char reference from user is treated as tainted, and it should be converted to untainted by doing proper checks before using it. This feature helps in identifying format string vulnerabilities.

Advantages: Since Splint is a static analysis tool, it helps to identify potential vulnerabilities even before compilation. It is fast and effortless to run similar to a compiler, but does lot more analysis than a compiler to identify potential vulnerabilities. Splint considers all program flows at procedure level unlike run time checkers, where program flows only allowed by test cases are considered. This attribute allows splint to analyze, detect and pin point potential vulnerabilities in code. Splint can detect memory leaks by ensuring all references complies its obligation to release storage. Splint enforces rigid coding guidelines, that helps programmer to write error free code. Also, Splint doesn't introduce any additional performance overhead (due to bounds checking) unlike run time checkers, where run time overheads are introduced. Annotated code provide more information about state of each procedure and makes understanding the code easier. Splint also have provisions for user defined checks, which allows splint to accommodate varied user requirements.



Ref	Base	Bound	Key	Lock addr
p	0x50	0x54	42	0xa0
q	0x70	0x74	43	0xa4

Fig. 1: Lock-key and base-bound in SoftBound+CETS

B. SoftBound+CETS

SoftBound+CETS (Compiler Enforced Temporal Safety) are compile time transformations for enforcing memory safety in C. While SoftBound detects spatial memory errors, CETS detects temporal memory errors. The rest of the section details key design and techniques used in SoftBound+CETS for detecting memory errors.

Spatial Errors: SoftBound associates every pointer with a base (b) and a bound (e) which is essentially the valid accessible memory region by the pointer. When a pointer tries to dereference a memory location, SoftBound checks if the pointer falls in-between base and bound linked with it to detect spatial memory violations. An exception is thrown if the pointer points outside the scope of b and e.

Temporal Errors: CETS augments every pointer with two metadata (1) allocation identifier known as *key* and (2) *lock* that is reference to a memory region called lock location [4]. The pointer is valid only when pointer's *key* matches *lock* location value. The visualization of the conceptual memory in SoftBound+CETS is shown in figure 1.

When a memory is deallocated, CETS changes the value at the lock location to `INVALID_KEY` (0), to avoid pointers having stale keys dereferencing the memory. A 64-bit global counter `next_key` is used for assigning the next unique key value. For preventing other temporal safety violations like double free (deallocating same memory twice), invalid free (freeing an address not returned by `malloc`), CETS maintains a hash map, mapping the pointer's *key value* to *freeable address* (address returned by `malloc`). Entry is added to the map whenever `malloc` is called. On deallocation, if this map does not have the pointer's *key*, it either means that memory is already deallocated or the program is trying to deallocate memory that was never allocated in the first place! This causes an exception to be raised and the program terminates. If the key is present, then that entry is removed from map and the corresponding memory is deallocated.

Metadata Propagation: As discussed above, SoftBound+CETS maintains metadata for all pointers to identify any spatial or temporal memory errors. This metadata is propagated along with pointers and checked on pointer dereferences. The following subsections describe how this metadata is propagated.

Pointer assignment and pointer arithmetic: Any pointer assignment (e.g. `p = q`) or pointer arithmetic (e.g. `p + 6`) will end up as pointer within that object. Pointer that doesn't point within its associated object after pointer operation is considered as violation and will not be dereferenced. The newly assigned pointer also has the same base, bound, lock and key as that of parent pointer.

Function calls: When a function returns pointers or when pointers are passed as arguments to functions, SoftBound and CETS transform every function call and declaration to include additional arguments. For every pointer argument, the metadata (base, bound, key, lock) arguments are appended to function's argument list.

Casts: SoftBound's disjoint metadata, unlike prior approaches of inline metadata, provides protection in the presence of casts [5]. The prior inline metadata methods have either not allowed the use of arbitrary type casts in the program or added additional structures to safeguard metadata from being overwritten by program operations [6]. CETS can only ensures that a pointer dereference happens only for allocated memory that are not stale, but does not give information about the type of that memory location [4].

Advantages: As SoftBound+CETS is compile time transformation, the source code requires no new annotations (like Splint) or rewrites (like cyclone). This allows for memory safe libraries to be built from existing libraries and makes it easier to adapt for existing projects. SoftBound+CETS's per-pointer metadata is maintained in a disjoint shadow memory. Hence, normal program memory operations cannot corrupt the metadata making it secure. It has support for arbitrary casts unlike other run-time checkers, where it is not supported. SoftBound+CETS has run-time information of program and can detect many memory violations that static analysis tool simply cannot. Once memory errors are detected SoftBound+CETS terminates the program, making it particularly suitable for security critical applications.

III. Benchmark Evaluation

This section describes about evaluation of Splint and SoftBound+CETS. The purpose of this benchmarking is two folds. (1) To show which among the two tools is efficient in detecting memory violations. (2) To estimate run-time and memory overheads introduced by these tools. The source code and setup details for bench marking are given in [7].

A. Effectiveness in Detecting Vulnerabilities

To evaluate the effectiveness of the two tools, Splint and SoftBound+CETS were applied to programs with security violations as listed in table I. The detailed explanation for the vulnerabilities are given [8], [9] and [10]. From Table I, we see that Splint detects more vulnerabilities than SoftBound+CETS. This is because, Splint considers program flows at procedural level to seek out coding flaws. In contrast, SoftBound+CETS detects only memory corruption at run-time caused by out-of-bound writes, dangling pointers, double frees and invalid frees.

TABLE I: Comparison of vulnerabilities detected

SN	Memory Violations	Violation Detected	
		Splint	SB+CETS
01	Buffer overflow	yes	yes
02	Use after free (dangling pointers)	yes	yes
03	Memory leak	yes	no
04	Double free	yes	yes
05	Undefined memory access	yes	yes
06	Stack variable reference	yes	yes
08	Buffer over-read	yes	yes
09	Null pointer dereference	yes	yes
10	Invalid memory free	no	yes
11	Unwanted aliasing	yes	no
12	Format string	yes	no

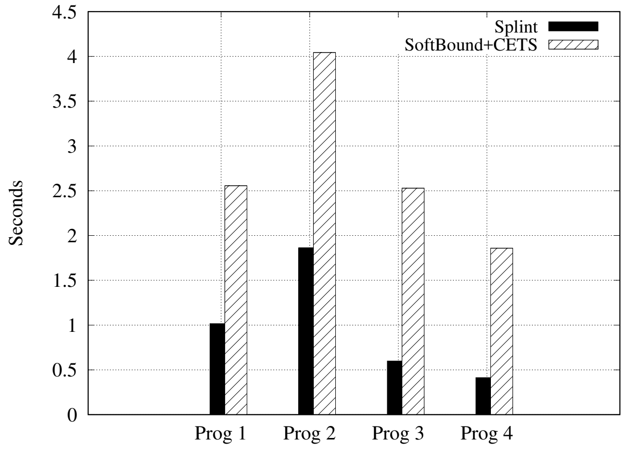


Fig. 2: Run-time comparison of pointer operations

B. Performance Overhead Evaluation

Benchmark: For evaluating performance overhead due to pointer operations, programs containing multiple pointer operations like pointer arithmetic, heap and stack allocation/deallocation were used [7]. The overhead introduced in terms of run-time and memory as a result of pointer operations, were evaluated for both tools.

Run-time overhead: The run-time of programs (having only pointer operations) compiled using Splint and SoftBound+CETS are given in figure 2. It can be observed that SoftBound+CETS has high run-time for pointer operations in contrast to Splint (average 120-350% higher). This can be attributed to (1) checking of bounds for buffer (SoftBound) and lock-key match (CETS), (2) frequent meta data accesses, which depends on frequency of pointer operations. This overhead is introduced due to compile-time transformation of source code by SoftBound+CETS. Splint does no such transformation, hence no run-time overhead is introduced.

Memory Usage: The memory usage of one of the benchmark program during execution (with repeated heap allocation/deallocation) is given for both Splint and SoftBound+CETS in figure 3. The fluctuations in graph signifies memory allocation (peak) and deallocation (bottom).

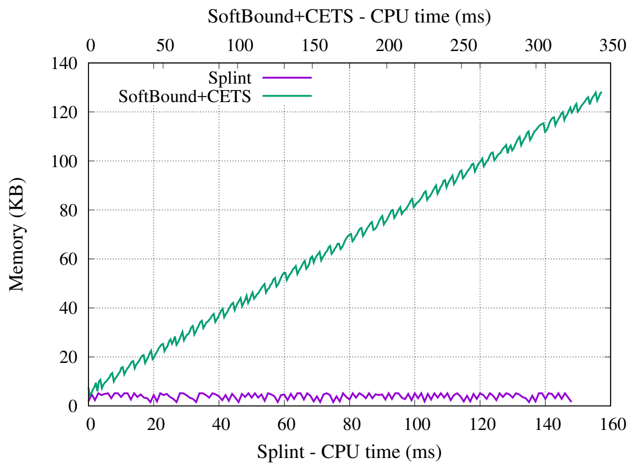


Fig. 3: Memory usage for program with pointer operations

In SoftBound+CETS, it can be seen that memory usage steadily increases throughout the run-time. This is because, CETS maintains a metadata table where for every memory allocation, an entry is made (lock and key) as illustrated in figure 1. But on memory deallocation, only the lock value is set to `INVALID_KEY` without removing the entry resulting in stale keys. This meta data table, introduces a memory overhead of few hundreds of KBs at the worst case. (program with large number of pointers) In contrast, memory usage is nearly constant throughout the run-time in Splint as it does not store any such meta data.

C. *Result*

From table I, it is seen that Splint detects more vulnerabilities than SoftBound+CETS. Also, the run-time overhead introduced by SoftBound+CETS is of serious concern for many applications as it greatly reduces performance of C. Although the memory overhead introduced by SoftBound+CETS (few KBs) is negligible for normal applications, it can be expensive for programs running in embedded system where even few KBs of memory is considered costly. Thus, Splint outperforms SoftBound+CETS in terms of both performance and effectiveness in vulnerability detection.

IV. Limitations and Recommendations

A. *Splint*

The main disadvantage of Splint is the amount of effort needed to annotate the program. This is especially cumbersome if annotations has to be done to existing to code that has thousands of lines. Another major issue, is the huge number of false positives and false negatives generated. Although these spurious warnings can be suppressed using flags, they delay development process. These limitations can be overcome by automating annotation process by combining run-time information along with Splint's static analysis. This immensely increases the usability factor of the tool.

Often, bounds of loop has to be estimated to determine buffer overflows and over-reads. Splint evaluates bounds using simple loop heuristics that is devised using human observation. In practice, these heuristics are not suitable for all loops and produces many false warnings. This can be overcome by implementing deeper analysis or providing extensions for user to provide information about loops to accurately determine loop bounds.

Splint detects vulnerabilities that are based on assumptions provided by programmer through annotations. This implies any mistake in assumption of variable or function state can cause splint to miss potential vulnerabilities. Also, it can lead to generation of large number false warnings. This limitation can be overcome by combining static analysis with run-time information obtained by running the program to reduce human errors.

B. *SoftBound+CETS*

One of the main limitations of SoftBound+CETS is its fail-stop approach (i.e the program halts when memory violation occurs). This is useful during development, but does not aid in real-life cases. The sudden termination of the program decreases its availability. Another problem with this approach is that, it allows hackers to conduct denial-of-service (DoS) attacks. To overcome this, SoftBound+CETS could provide support for alternate execution path for program whenever memory violations are detected.

Furthermore, bound checks and frequent metadata access in SoftBound+CETS incur some performance penalty. This limitation can be overcome by providing options to disable these checks on fragments of code where the programmer is certain there are no memory violations. This can give a sizable boost to the performance.

Another limitation of SoftBound+CETS is that it cannot detect vulnerabilities before deployment. In other words, SoftBound+CETS is not pro-active. Only when violation occurs during run-time, it can detect them. Static checks can be incorporated along with SoftBound+CETS to complement it. Any vulnerability missed by static checks can be caught during run-time.

V. Conclusion

In this report, design, techniques, advantages and disadvantages of Splint and SoftBound+CETS are discussed. Softbound+CETS requires no effort by the programmer to adapt, which makes it suitable for existing projects. Though it detects most of the memory errors, it is done only during the run time. It does not consider all execution path of the program, which make it limited in its capabilities. Also, the performance overhead introduced by them may be expensive for performance oriented applications.

Splint on the other hand, is a light weight static analysis tool, which enforces programmer to write code without vulnerabilities. It detects memory errors without introducing any overhead to the program which makes it suitable for performance oriented applications. Also, the extensible checks allow user to define customized checks that can greatly improve quality of the program. Although the effort required to adapt Splint to codes and libraries is cumbersome, the benefits of Splint outweigh the disadvantages. If techniques are developed to combine run time information with static analysis to automate annotation process, Splint can be a powerful tool in detecting vulnerabilities in code.

References

[1] D. Larochelle and D. Evans, *Splint Manual*, University of Virginia.
[2] ——, "Statically detecting likely buffer overflow vulnerabilities," *IEEE software*, 2001.
[3] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *IEEE software*, no. 1, pp. 42–51, 2002.
[4] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Cets: compiler enforced temporal safety for c," *IEEE software*, vol. 45, no. 8, pp. 31–40, 2010.
[5] ——, "Softbound: Highly compatible and complete spatial memory safety for c," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 245–258, 2009.
[6] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "Ccured: type-safe retrofitting of legacy software," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 3, pp. 477–526, 2005.
[7] "Memory safety benchmark," https://github.com/vijathanga/memory-safety, accessed: 2018-10-25.
[8] "How to avoid, find memory errors in your c/c++ code," https://www.cprogramming.com/tutorial/memory_debugging_parallel_inspector.html, accessed: 2018-10-25.
[9] "Memory safety," https://en.wikipedia.org/wiki/Memory_safety, accessed: 2018-10-25.
[10] "Memory safety analysis," http://www.semdesigns.com/Products/MemorySafety/, accessed: 2018-10-25.

TABLE II: Contributions of team members

Contributions	**Vijay**	**Vipul**	**Sonal**
Splint	Yes	Yes	Yes
Softbound+CETS	Yes	Yes	Yes
Benchmarking	Yes	Yes	Yes

Appendix

Experimental Setup

All benchmark tests were evaluated on

- OS - Ubuntu 64-bit
- RAM - 8 GB
- Processor - i7 7th gen (4 cores)
- Compiler - Clang version 3.4