LLVM Based Bug Detection

A comparison of CETS and Parfait

Sebastian Hunkeler, shunkele(at)hsr.ch

University of Applied Sciences Rapperswil, HSR
Department of Computer Science, MRU Software and Systems

Keywords: LLVM, pointer safety, bug detection, static analysis, CETS, Parfait

Acknowledgement

I would like to thank my advisor Peter Sommerlad for his attentive reading and many helpful comments. And of course many thanks to Yves Thrier for his valuable feedback.
1 Introduction

Programming languages that inherently use managed memory such as C# or Java have become very popular during the last years [2]. The high abstraction of the memory management and the ability to omit pointer arithmetic simplifies the programming work by factors [9]. As for Java, the only way to allocate memory is the use of the new operator. The only possibility to free memory is to rely on the garbage collector [19]. However, the usage of such high-level languages is not appropriate for all application domains because of the overhead introduced by the garbage collecting mechanism. Although garbage collection is considered a constant factor [12], it comprises a significant and non-deterministic runtime overhead.

Devices such as smart phones or embedded devices have limited hardware resources but require realtime constraints. Therefore, they mostly rely on system programming languages with explicit or deterministic resource management such as C or C++ for the system foundation. Both C and C++ allow the use of pointers which is a powerful and performant way of accessing memory. But handling raw pointers is error-prone and can lead to security flaws and system instability. In order to counter these problems tools for static analysis and runtime instrumentation can be adopted.

In this paper I will analyze and compare two bug detection tools for C which provide spatial and temporal pointer safety, respectively. These tools, namely Compiler-Enforced Temporal Safety for C (CETS) [14] and Parfait [6] are built on top of the Low Level Virtual Machine (LLVM) project, a collection of industrial strength compiler technologies (section 1.2). In order to compare CETS and Parfait I propose criteria for bug detection tools in section 1. In section 2 the motivation behind these projects is examined in detail and examples of concrete applications are suggested. In section 3, I present the main differences between the approaches and give pros and cons for each of them. I end with a conclusion in section 4.
1 Introduction

1.1 Related Work

The original papers for CETS [14] and Parfait [5, 13] provide a comprehensive overview on how these tools are implemented and how they perform. However, they do not furnish a wider perspective on how they could be adapted in the daily software development process. Furthermore the applicability for the different usage scenarios is left open.

1.2 The Role of LLVM

LLVM is a powerful set of modular, reusable compiler components [11]. LLVM emerged from the need of a modern, high-performant, extensible compiler architecture. It includes LLVM-GCC which is a drop-in replacement for the GNU Compiler Collection, replacing both the optimizer and code generator [11]. LLVM-GCC has been created because the original GCC lacks cross-file/link time optimization and it has an aging code base which makes it hard for changes. LLVM-GCC’s link time optimization (optimization level 4) brings a performance boost of 20% compared to the highest optimization level -O3 of the original GCC. In addition the compile time of LLVM-GCC is better on each optimization level compared to the original GCC [11].

Since runtime instrumentation tools for pointer safety can cause a performance decrease by factors up to 10 [14], efficiency is an important concern. Besides the better performance of LLVM-GCC, the modularity and reusability of the compiler components make it easy to satisfy the optimization needs of these tools. Because the LLVM optimizer operates on an intermediate representation, code injection by code analysis tools can be performed independent of the Instruction Set Architecture (ISA) [14].

Using LLVM for compiler-based code injection has the significant advantage that a dual optimization of the code can be performed easily. Consider the following example:

```c
#include <stdlib.h>
```
int main(int argc, const char* argv[])
{
    int* foo = malloc(sizeof(int));
    foo = 1;
    checkPointer(foo);

    int* bar = malloc(sizeof(int));
    checkPointer(bar);

    return 0;
}

In the example code a function checkPointer() was injected during compilation which performs some analysis on each pointer in the program. Before the code injection, the variable bar was never read in the program and would therefore have been removed by the optimizer. But since we introduced additional code that analyzes every single pointer, this variable is indeed read and thus not removed by the optimizer. A solution for this problem is to conduct a dual optimization as mentioned above. This will remove the unused variable bar in the first phase, then inject the specified code and re-optimize again the original code combined with the injected code (Figure 1 on page 3).

It is also necessary to note that LLVM aims to provide better tools for source level analysis and improved debugging information. This is especially valuable when the injected code should communicate with the debugging environment (see section 2.2.2 on page 12).

1.3 Criteria for Source Code Analysis Tools

For analyzing source code in order to find bugs and anomalies, there are different approaches. Depending on the information to be obtained, static analysis, dynamic analysis or a combination of them is appropriate. Static analysis is a way of testing software without running it. Test results are obtained by performing formal checks. Since it is performed on source code, it is considered a white-box test [10]. For dynamic analysis in contrast, the software is analyzed during runtime. Some tools for dynamic analysis (e.g. profiling tools) use instrumentation (see section 2.1) to gather information about a specific state of the running application [18].

Like other testing mechanisms, static and dynamic analysis of software try to be complete, correct and performant. Unfortunately completeness cannot be achieved because of Rice’s theorem [16]. Therefore false negatives may exist. Absolute correctness may be achieved, depending on the complexity of the analyzed problem. For simple bug types no false positives should be reported in the test results. CETS and Parfait in each case tackle a different class of software problems and serve different purposes. CETS focuses on temporal pointer safety. It solves the problem of temporal memory access violations completely. This means, that 100% of the violations can be detected. But since CETS can be used during the runtime of a software, performance and compatibility with third-party libraries is important. The biggest concern for CETS, however, is
correctness, because it is meant to be applied throughout a complete software system. Parfait on the other hand is rather adequate for partial application during the development cycle. Since it can detect a variety of bugs, it is also likely to generate false positives which must be falsified manually. If these tools are used during software development, compile time is a significant factor as well.

The need for such tools increased during the last years, because nowadays almost every computer system is connected to the internet. Formerly closed systems such as banking systems are now exposed to the public because of e-banking services. Even our mobile phones are connected to the internet. Bugs in software can lead to data loss or unauthorized access to such systems.

Safety in sense of robust error handling is especially an important concern for software supporting our lifes. This kind of software is often installed on embedded devices such as heart pacemakers and needs to be 100% reliable.

1.4 Faulty Pointers

Faulty pointers are pointers which do not point to a valid address. There are several kinds of faulty pointers which are explained in the following.

Wild Pointers Wild pointers are pointers which have no implicit initialization and are not initialized prior to first use. Therefore they can point to arbitrary memory locations. In such a case the behavior for accessing them is undefined.

Dangling Pointers Another type of faulty pointers are dangling pointers. As for C, deleting an allocated memory range (AMR) does not change the associated pointers. Once an AMR is deleted, the associated pointers point to an invalid memory location and become dangling. A pointer can become dangling in the following situations:

1. The AMR on the heap the pointer points to has been freed explicitly
2. The AMR the pointer points to has implicitly been deleted after the associated stack frame was destroyed (e.g. on function return)

Out-of-Bounds Pointers Out-of-bounds pointers point to an address beyond the defined bounds of an AMR. These pointers are derived from a pointer returned by a memory allocation function. If pointer arithmetic is applied without applying proper bounds checking, a pointer can become faulty because it points to an address which is not within the allocated memory range anymore.

Null Pointers Null pointers are intended to indicate that a pointer points “nowhere”. Pointers can be set to null on purpose hence they are no faulty pointers. However, accessing them cause an error and is therefore a major bug.
1.4.1 Spatial vs Temporal Pointer Safety

Because CETS focuses on dangling pointers and the initial implementation of Parfait focuses on out-of-bounds pointers these two types need to be distinguished. In the following temporal pointer safety refers to safety for dangling pointers and spatial pointer safety refers to safety for out-of-bounds pointers.

Temporal Pointer Safety A temporal safety violation occurs whenever a pointer variable initially pointed to a valid address but eventually points to an invalid address due to freeing and possibly reallocating memory over time. This may either be because of explicit calls to malloc(), mmap() or free() and munmap() or because the corresponding stack frame was destroyed. Consider the following example:

```c
int* foo = malloc(sizeof(int));
*foo = 12;
int* bar = foo;
printf("foo: %p : %d\n",foo,*foo);
printf("bar: %p : %d\n",bar,*bar);
free(foo);
//bar is now a dangling pointer to a freed region

int* foo2 = malloc(sizeof(int));
//bar may now be a dangling pointer to a reallocated region
//depending on the malloc implementation
*foo2 = 24;
printf("foo2: %p : %d\n",foo2,*foo2);
printf("bar: %p : %d\n",bar,*bar);
free(foo2);
```

In this code snippet we can find two different types of dangling pointers. At line 7 bar is a dangling pointer because it points to a freed region. At line 10 bar may turn into another dangling pointer because it can point to a region which has been reallocated. In this case the application doesn’t crash but fails silently. The output of the above code produced on my computer running OS X is as follows:

```
foo: 0x100100080 : 12
bar: 0x100100080 : 12
foo2: 0x100100080 : 24
bar: 0x100100080 : 24
```

As we can see, in this case at line 10, bar becomes accidentally a dangling pointer to a reallocated region. foo2 has been assigned the same memory address as foo had before freeing it. Apparently bar again points to a valid address and yields the value of foo2. However, the behaviour of such a code is undefined.

Spatial Pointer Safety Spatial pointer safety expresses that a pointer should not point beyond the intended bounds. When using pointer arithmetics with-
out doing boundary checks, such a memory violation can happen very quickly. Consider the following example of a buffer overflow:

```c
char text[] = "Foo Bar";
char buffer1[4], buffer2[4];

printf("buffer1: %p - %p\n", buffer1, &buffer1[3]);
printf("buffer2: %p - %p\n", buffer2, &buffer2[3]);

int i;
for(i=0;i<sizeof(text);++i)
  buffer2[i] = text[i];

printf("%3.3s %3.3s\n", buffer1, buffer2);
```

In this code snippet two buffers of four bytes are allocated. Then in the for loop, data from the text array is written to the memory starting at the address of buffer2. Since for x86 architectures, stack memory addresses go in decreasing order we need to start at the address of buffer2 in order to overwrite buffer1. Eventually we write the data of the text array across the bounds of buffer2 and commit a spatial memory violation.

| buffer1: 0x7fff5fbff8a0 - 0x7fff5fbff8a3 |
| buffer2: 0x7fff5fbff890 - 0x7fff5fbff893 |

As we can see in the output, buffer1 contains the string “Bar” and buffer2 contains “Foo”. In order to make this example work, I inserted 13 dummy characters in the text array. This is because of the stack alignment on my system.

There are known system functions such as `strcpy()` which lack a bound check and should therefore not be used. Detecting the presence of these functions could trivially be done using a text search tool such as `grep`. But in a situation as in the above code snippet further investigation is required in order to detect memory violations.

## 2 Analysis Tools

Both CETS and Parfait serve for the detection of bugs in C source code. C is an unmanaged programming language which allows arbitrary memory access with pointers. Using pointer arithmetic is often error-prone and probably the largest source of bugs. The purpose of pointer safety is to detect erroneous memory access through faulty pointers.

### 2.1 Operating Modes

Because CETS and Parfait tackle different problems, they use different modes of operation. Dangling pointers can appear during runtime depending on the program inputs which are unpredictable. Therefore CETS cannot completely detect dangling pointers through static analysis. Instead it uses compiler-assisted
instrumentation. CETS patches the source code and inserts checks during compilation (see figure 2) which then detect dangling pointers during runtime. Since the information is gathered during runtime this can be regarded as a kind of dynamic program analysis [18].

![Compiler-Assisted Instrumentation](image)

**Fig. 2:** Compiler-Assisted Instrumentation

Parfait in contrast performs only static analysis. Through a formal analysis of the source code, potential bugs can be found. Some of these bugs can be verified automatically while others may need a manual review. Parfait uses a compiler-assisted static analysis approach. Thus the source code is being analysed during compilation (see figure 3). The output of this analysis can be presented to the developer for review. This allows the developer to find and correct existing bugs.

![Compiler-Assisted Static Analysis](image)

**Fig. 3:** Compiler-Assisted Static Analysis

### 2.2 CETS

CETS is a powerful tool which allows to detect all dangling pointers caused by temporal violations. CETS’ guarantee of temporal pointer safety only applies provided that spatial safety preexists. This may be accomplished through the usage of a tool that ensures complete spatial safety such as SoftBound [13]. The reason for this is that CETS maintains shadow space metadata for the pointers.
Since this metadata may be corrupted because of spatial memory violation, complete spatial pointer safety is a prerequisite.

Consider the example in figure 4: If the data in the section labelled “metadata shadowspace” is changed by an out-of-bounds write operation in section “application data” the metadata can become tainted.

```c
int* ptr1 = malloc(sizeof(int));
int* ptr2 = ptr1;
char* ptr3 = malloc(1);
...
```

Fig. 4: CETS Metadata Shadowspace Example

The actual temporal pointer checking of CETS is performed during runtime. For each pointer dereference a check is carried out that determines if the dereference operation for the address to which the pointer refers is allowed (see section 2.2.1). If the operation is invalid the process receives a SIGABRT signal using the `abort()` system call. This changes the behavior of accessing a dangling pointer from undefined to defined. The default behavior is namely terminating the current process. While it can prevent data corruption and arbitrary application crashes, it is not always a safe option to use `abort()`. This mechanism can be leveraged to higher level purposes through installing a signal handler for SIGABRT. It could either be used for debugging, printing error messages or for error recovery. It could be used as a unit of mitigation [8] and enable the system to recover in situations where a subsystem fails. A security relevant webservice could also switch to a deny-all state when detecting a dangling pointer. This keeps the system in a well-defined state. Although a denial of service attack could then be effected, it would prevent data corruption and leakage of highly sensitive information.

### 2.2.1 Functionality Example of CETS

In order to detect temporal safety violations CETS performs three fundamental steps:

1. On each memory allocation store metadata about the allocated region
2. Before dereferencing a pointer perform a validity check
   - (a) If the pointer is valid continue
(b) If the pointer is invalid call `abort()`

3. On each deletion of allocated memory track the deletion by updating the corresponding metadata

The metadata of CETS contains a key and a lock address for each pointer (figure 4). It stores them in a disjoint metadata space because the program’s memory layout should remain unchanged to enforce compatibility with existing source code and precompiled libraries. In this case temporal safety is guaranteed even in the presence of arbitrary casts and reallocations.

In order to demonstrate how CETS works we have a look at a simplified version of the example of section 1.4.1 and show how the detection mechanism works even in the presence of a dangling pointer to a reallocated region. While the example source code is working C code, the instrumented CETS code (highlighted using a gray background) is pseudo code to simplify the example. For simplicity the instrumented code for detecting double frees and frees to pointers not returned by `malloc()` is left out as well.

Before CETS is instrumented the code looks as follows:

```c
int* foo = malloc(sizeof(int));
*foo = 12;
int* bar = foo;
free(foo);
int* foo2 = malloc(sizeof(int));
*foo2 = 24;
printf("bar: %p : %d\n",bar,*bar);
```

On each memory allocation a new unique key `ptr_key` is generated which is never reused later on. Also a lock address `ptr_lock_addr` is assigned to the corresponding pointer. The lock value is set to the value of `ptr_key`. After applying this step for the two allocations on lines 1 and 5 the code for these lines is as follows:

```c
int* foo = malloc(sizeof(int));
ptr_key = next_key++; //return the next unique key e.g. 1
ptr_lock_addr = allocate_lock(); //lock for memory region
*(ptr_lock_addr) = ptr_key; //assign ptr_key to lock value
store(foo,ptr_key,ptr_lock_addr); //store metadata

int* foo2 = malloc(sizeof(int));
ptr_key = next_key++; //return the next unique key e.g. 2
ptr_lock_addr = allocate_lock(); //lock for memory region
*(ptr_lock_addr) = ptr_key; //assign ptr_key to lock value
store(foo2,ptr_key,ptr_lock_addr); //store metadata
```

Since `bar` on line 3 points to the same memory region as `foo` the same key and lock address is assigned:

```c
int* bar = foo;
```
ptr_key = lookup(foo)->key;
ptr_lock_addr = lookup(foo)->lock_addr;
store(bar,ptr_key,ptr_lock_addr);

On dereference of a pointer it checks if the key still matches the lock value. If they don't match, a dangling pointer is detected. After applying these checks to the three dereference operations on lines 2, 6 and 7 the code becomes the following:

ptr_key = lookup(foo)->key;
ptr_lock_addr = lookup(foo)->lock_addr;
if (ptr_key != *ptr_lock_addr) abort();
*foo = 12;

ptr_key = lookup(foo2)->key;
ptr_lock_addr = lookup(foo2)->lock_addr;
if (ptr_key != *ptr_lock_addr) abort();
*foo2 = 24;

ptr_key = lookup(bar)->key;
ptr_lock_addr = lookup(bar)->lock_addr;
if (ptr_key != *ptr_lock_addr) abort();
printf("bar: %p : %d\n",bar,*bar);

On deletion of an allocated memory region the value of the lock address corresponding to the pointer is set to INVALID_KEY and the lock is deallocated. After applying this rule to the deletion on line 4 the code is as follows:

free(foo);
ptr_lock_addr = lookup(foo)->lock_addr;
*ptr_lock_addr = INVALID_KEY;
deallocate_lock(ptr_lock_addr);

On line 7 the pointer bar is accessed and therefore checked for its validity. This is done through a comparison of the ptr_key and the lock value. Since bar is a dangling pointer, the keys for the lock values must not match. This can be seen in the following table:

<table>
<thead>
<tr>
<th>Line number</th>
<th>ptr</th>
<th>ptr_key</th>
<th>ptr_lock_addr</th>
<th>lock address</th>
<th>lock value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>foo</td>
<td>1</td>
<td>0xABCD</td>
<td>0xABCD</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>foo</td>
<td>1</td>
<td>0xABCD</td>
<td>0xABCD</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>bar</td>
<td>1</td>
<td>0xABCD</td>
<td>0xABCD</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>foo</td>
<td>1</td>
<td>0xABCD</td>
<td>0xABCD</td>
<td>INVALID_KEY</td>
</tr>
<tr>
<td></td>
<td>bar</td>
<td>1</td>
<td>0xABCD</td>
<td>0xABCD</td>
<td>INVALID_KEY</td>
</tr>
<tr>
<td>5</td>
<td>foo</td>
<td>1</td>
<td>0xABCD</td>
<td>0xABCD</td>
<td>INVALID_KEY</td>
</tr>
<tr>
<td></td>
<td>bar</td>
<td>1</td>
<td>0xABCD</td>
<td>0xABCD</td>
<td>INVALID_KEY</td>
</tr>
<tr>
<td></td>
<td>foo2</td>
<td>2</td>
<td>0xCDEF</td>
<td>0xCDEF</td>
<td>2</td>
</tr>
</tbody>
</table>
For each instruction which causes the metadata to change, all present pointers together with their associated metadata are listed. When accessing bar on line 7 the state of the metadata is still the same as on line 5. Thus the ptr_key of bar is set to 1 and the lock value is set to INVALID_KEY. Since 1 is not equal to INVALID_KEY the check evaluates to false and the dangling pointer has been detected.

This example only shows how to detect one type of dangling pointers. In order to detect all dangling pointers CETS also handles other scenarios which are not discussed in this paper.

2.2.2 Applicability for Debugging

One application of CETS is the use of the tool as a debugging aid. A very useful scenario which I always come across when developing iPhone software, is debugging reference counting errors of the Cocoa Touch framework. On the iPhone there is no garbage collector available thus you have to use reference counting which can be very error-prone. For non-void methods it is not clear if the caller or the callee increases the reference counter of an object by retaining it. Objects returned by convenience factory methods must be retained while explicitly allocated objects are retained automatically. The Objective-C memory model can be very confusing in such scenarios. There are often errors regarding reference counting which lead to program crashes. Once the program is crashed you have to step through the code and log retain counts in order to find the culprit. What actually happens in this case is a temporal memory violation. In the following I will show how CETS’ technique can be augmented to assist finding reference-counting bugs in the source code.

When developing applications for iOS devices with Objective-C, you will realize that the manual reference counting memory model is very error-prone. Objects get automatically deallocated once their reference count is zero. Each time one calls the release method, the counter is decremented. If there is a semantic error in your code and the release method is called once too often the program crashes. In fact this is because objc_msgSend(ptr, sel_getUid("release")) is called with the dangling pointer ptr as its argument. In version 3 of Apple’s Xcode integrated development environment (IDE), a developer had no indication for where to start looking for the erroneous release calls. This has improved since version 4 which uses LLVM and the Clang compiler frontend. The IDE now points to the method where the crash actually happened. But it remains to the user to track down where the multiple calls to release were carried out and on which code line the last call to release, which led to the crash is located.

I implemented a conceptual prototype which builds on top of CETS’ technique, enabling an automatic tracking of such calls. As soon as a dangling pointer is detected by CETS, the information could be used to activate the debugger and visualize where the affected calls are located. The visualization part would have to be adapted by Apple, because the Xcode source code is not publicly available. I tried two slightly different approaches. The first requires
a preprocessing step that replaces the calls to the original release with a call to an upstream release method. The second method, which is not further discussed in this paper, doesn't require this replacement of the release call. But it would require access to debugging information at runtime which is more complex than the first approach.

In order to simplify the preprocessing step of the release method call replacement, I implemented the upstream release method as a category method [1], thus the target method gets mixed into the root class NSObject.

```objective-c
@implementation NSObject (override)
- (void) debugRelease:(id)key info:(NSString*)info {
    [NSSafeObject saveReleaseLocation:key withInfo:info
     retainCount:[self retainCount]];[
self release]; //Call original implementation
}
@end
```

The NSSafeObject class implements the storage for the metadata. It comprises a method to save the metadata for the release calls, a method for getting the release locations as well as the free location. The free location is the location of the release call which caused the reference count to become zero. To distinguish between release locations and free locations two separate data structures are used. I used the NSMutableDictionary class for these collections. This has the advantage that metadata look-ups can be performed in O(1).

```objective-c
@implementation NSSafeObject

static NSMutableDictionary* freeLocations;
static NSMutableDictionary* releaseLocations;
...
+ (void) saveReleaseLocation:(id) key
    withInfo:(NSString*)info retainCount:(NSUInteger)count{
    if (count==1) {
        [freeLocations setObject:info forKey:key];
        return;
    }[NSMutableArray* locations = [releaseLOCations objectForKey:key];
    if (locations=nil) {
        locations = [(NSMutableArray alloc) initWithCapacity:DEFAULT_SIZE];
        [releaseLocations setObject:locations forKey:key];
        [locations release];
    }[locations addObject:info];
}
+ (NSString*) getFreeLocation:(id) key{
    return [freeLocations objectForKey:key];
}
+ (NSArray* getReleaseLocations:(id)key{
```
NSMutableArray *locations = [releaseLocations objectForKey:key];
return [NSArray arrayWithArray:locations];

...
Fig. 5: Xcode Dangling Pointer Visualization Mock-up

Xcode’s built-in static analyzer is able to detect dangling pointers in trivial cases. If a pointer to an object does not escape from the method it was allocated in and the object gets deallocated inside the same method, then the static analyzer can detect an access to this dangling pointer. However, in practice this constellation is rarely the case. In more complex situations the approach I explained in this section can serve as a very useful debugging aid which may alleviate the bug-finding process significantly.

2.3 Parfait

In contrast to CETS which is about temporal memory safety, Parfait aims to detect also higher layered bugs such as SQL injection, cross-site scripting or weak passwords [6]. It is a bug checking framework designed for scalability and precision. Since Parfait enforces security constraints, it has a preprocessing analysis pass, creating a reachability graph. With this graph, Parfait determines which parts of the code are reachable by user inputs and therefore need to be analyzed in detail. In this case some program analysis such as constant propagation can be omitted since constant values cannot stem from user-input dependencies [6]. Scalability is a major concern for Parfait as well, because it should be integrated into a day-to-day software development process to evolve its full potential [5]. Therefore the code analyses should be fast. In order to achieve this requirements, a demand-driven, parallelized multilayered approach is used.
The different analysis layers vary in complexity and expense. The applied layers are ordered by increasing complexity. This ensures that the cheapest program analysis (PA) capable of detecting a bug is adopted [5]. The analysis process can be stopped as soon as a program analysis can either verify or falsify a potential bug.

Only small slices of code around a potential buggy statement are analyzed [6] instead of analyzing the whole program which takes much more time. For each bug type Parfait assembles a list of potentially buggy statements [6]. The framework processes this list with the multilayer approach and refines the test results. Eventually, Parfait produces a list of real bugs, no bugs, and potential bugs. The potential bug list needs to be verified manually. The algorithm of the Parfait framework operates as follows:

1. Set up worklist for specific bug type (e.g. buffer overflow)
2. Populate a set with all statements that potentially can cause the bug
3. Iterate over the program analysis in the ensemble in ascending order (increasing complexity)
   Analyze the statements in the worklist. For each statement:
   (a) Report real bug / non-bug
   (b) Retain potential bug in worklist for further analysis by later analysis
4. Remaining statements in worklist are marked as potential bugs and need to be verified manually

---

**Fig. 6: Parfait Architecture (Source: [6])**
This algorithm can be parallelized on two levels. “The first level is the various worklists for specific bugs (e.g. buffer overflows, string vulnerabilities, integer overflows, etc.) and the second level is for statements in a specific worklist [6].” Since both the worklists and the statements have no dependencies, they can be processed individually and in parallel.

2.3.1 Functionality Example of Parfait

The initial implementation of Parfait focuses on buffer overflows for read and write operations in arrays. This implementation comprises a potential-bug-locator which produces a list of all locations in a C program where these buffer overflows may exist [6]. Parfait uses constant propagation and partial evaluation to detect the buffer overflows in these locations. The partial evaluation code is run using the LLVM interpreter [6]. In order to show how constant propagation and partial evaluation works, a slightly modified version of the second example of section 1.4.1 is given:

```c
char text[] = "Foo Bar";
char buffer1[4], buffer2[4];

int i, n = sizeof(text);
for(i=0;i<n;++i)
    buffer2[i] = text[i];
printf("Last char of text is: %c",text[n]);
```

**Constant Propagation**  Constant propagation is the process of substituting the values of known constants in expressions [6]. In our example the array `text` has a constant size `n = 20` (including the null termination). The substitution of the constant values the code leads to the following code snippet:

```c
char text[] = "Foo Bar";
char buffer1[4], buffer2[4];

int i, n = 20;
for(i=0;i<20;++i)
    buffer2[i] = text[i];
printf("Last char of text is: %c",text[20]);
```

In C an array with `n` items is indexed from 0 to `n-1`. In this case the last defined index access is `n - 1` equals 19. The constant access to the 20th index of `text` on line 7 can be detected as an out-of-bounds array read since the index 20 is greater than 19 and therefore is undefined. Because the access to `buffer2` on line 6 is not constant, constant propagation is not sufficient. Instead Parfait uses a partial evaluation approach.

**Partial Evaluation**  Partial evaluation is a program transformation technique which allows to evaluate only a small part of source code. “For buffer overflow checking, partial evaluation can be used when analyzing a loop that accesses
an array and has a constant number of iterations.” [6]. In this case bounds-checking code for the index variable \( i \) can be inserted and evaluated using the LLVM interpreter. The corresponding code snippet for line 6 of the example is as follows:

```c
for (i=0;i<20;++i){
    if (i < 0 || i > 3) return true; // Check for buffer2 access
    if (i < 0 || i > 19) return true; // Check for text access
}
return false;
```

Since the check for the buffer2 array write is more restrictive than the check for the text array read, the latter one could be removed during an optimization phase. If this snippet is run using the LLVM interpreter it returns true indicating that the buffer overflow bug has been detected.

### 3 Comparison

#### Detectable Bugs
CETS detects all temporal safety violations and coupled with SoftBound it detects all spatial safety violations in C source code as well. This means that it can detect all bugs that are based on faulty pointers such as buffer overflows or dangling pointers. The following list shows the distinction between the detectable temporal and spatial safety violations:

<table>
<thead>
<tr>
<th>Temporal safety violations (CETS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dangling pointer to freed region</td>
</tr>
<tr>
<td>Dangling pointer to stack frame</td>
</tr>
<tr>
<td>Dangling pointer to reallocated region</td>
</tr>
<tr>
<td>Calling free to address not returned by malloc</td>
</tr>
<tr>
<td>Double freeing a pointer returned by malloc</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Spatial safety violations (SoftBound)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write/read of memory location that is out of bounds of an array</td>
</tr>
<tr>
<td>Write/read of memory location that is not returned by malloc</td>
</tr>
</tbody>
</table>

While the initial implementation of Parfait focuses on buffer overflows in C source code, the following example issues could be detected in future versions [6]:
### Input Validation and Representation Category

- Buffer overflows
- Command injection
- Cross-site scripting
- Format string
- Integer overflow
- SQL injection

### Timing and State Category

- Deadlocks
- Race conditions

### Security Features Category

- Insecure randomness
- Least privilege violation
- Missing access control
- Password management
- Privacy violation

### API Abuse Category

- Dangerous functions that cannot be used safely
- Directory restrictions
- Heap inspection
- Misused language or operating system features

---

**Demand-Driven vs Complete Analysis**

A complete analysis of source code is obviously better than a partial or demand-driven one. But a complete analysis is not always appropriate. For example, in large code bases of millions of lines of code a complete analysis can take weeks to finish. Therefore a major goal of Parfait is to preserve the applicability for large code bases through a partial analysis. Since the time-consuming checks of CETS are carried out during runtime, a complete instrumentation during compilation is accurate.

**Applicability to Software Development Stages**

Although CETS can be instrumented during the development process for testing, the application of CETS during runtime is especially valuable for security and reliability purposes (see section 2.2 on page 8). Parfait on the other hand must inevitably be used during development. The application flow of Parfait is to first perform an analysis of the code. The test results are used to fix the bugs and re-run the analysis to make sure the bugs are properly fixed.

**IDE Integration**

CETS as well as Parfait are built on top of LLVM. Thus they integrate well with LLVM-GCC, which is a drop in replacement for the GNU version of GCC. Therefore CETS and Parfait can generally be used everywhere where GCC / LLVM-GCC can be used. Both CETS and Parfait are not yet publicly available hence little information is available on how to use these tools.
Since CETS is derived from the code of SoftBound [13] it is very likely to operate in a similar way. SoftBound is built as a shared library which is deployed to the LLVM-GCC library directory. The source is compiled using LLVM-GCC with the -fsoftbound-hash flag set, which loads the SoftBound library and runs the instrumentation. If CETS operates the same way it could be easily integrated into an IDE such as Eclipse by editing the compiler settings for the target project. The gcc command would then be replaced with llvm-gcc or the LLVM-GCC version is symlinked to gcc. The textual output of CETS needs to be interpreted by an Eclipse plug-in for proper visualization.

Parfait is built as a standalone GCC compiler which does the necessary preprocessing and compilation. The command for calling the compiler from the command line is parfait-gcc. A separate command line tool named parfait is responsible for the actual analysis of the files produced by the Parfait compiler. The output of Parfait is textual and can therefore be parsed and visualized using a plugin for Eclipse or other IDEs.

**Customizability** Customizability of a tool for static analysis is mostly a good characteristic assumed that reasonable defaults exist. If they don’t exist, customization can be a tiresome and a time-consuming work. Depending on the complexity of the bugs to be detected, customization is not an optional task. CETS does not have any possibility for customization except modifying its source code. But since it focuses on one particular issue, there is no need for customization. Parfait on the other hand cannot be used without customizing it, because it enforces a demand-driven approach. A user must specify which bugs should be detected and which parts of the software should be analyzed. For specific bug types the analysis layers need to be specified and implemented. With this approach it is possible to customize Parfait to satisfy the specific needs of the developer.

**Summary** The following table shows a summary of the qualities and differences between CETS and Parfait.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>CETS</th>
<th>Parfait</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target programming language</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>Main goal</td>
<td>Complete temporal pointer safety</td>
<td>Precise bug detection designed for scalability</td>
</tr>
<tr>
<td>Operating mode</td>
<td>Instrumentation</td>
<td>Static analysis</td>
</tr>
<tr>
<td>Application extent</td>
<td>Throughout software</td>
<td>Selective</td>
</tr>
<tr>
<td>Development phases</td>
<td>Debug, release</td>
<td>Debug</td>
</tr>
<tr>
<td>Operation purpose</td>
<td>Testing/runtime stability and security</td>
<td>Testing, bugfixing aid</td>
</tr>
<tr>
<td>Detection coverage</td>
<td>Complete</td>
<td>Partial</td>
</tr>
<tr>
<td>Deployment</td>
<td>Compiler library for instrumentation</td>
<td>Standalone compiler + static analysis tool</td>
</tr>
<tr>
<td>IDE integration</td>
<td>Possible</td>
<td>Possible</td>
</tr>
</tbody>
</table>
4 Conclusion

If access protection and reliability of your C based software is your largest ambition and if you can afford the runtime overhead of 48% respectively 116% when coupled with SoftBound, CETS is the tool of choice. Together with a spatial memory safety tool, it preserves a deterministic state of your software and ensures complete memory safety for your software. Thus as a last resort when detecting a dangling pointer, a webservice software such as an e-payment webservice could reject all incoming requests and prevent unauthorized access. In this case it could prevent the leakage of sensitive credit card information. However, CETS only detects low-level issues and is not able to detect more complex security violations.

Parfait on the other hand can detect more abstract issues and can be extended to detect arbitrary bugs according to the needs of a developer. Nevertheless using Parfait, the detected bugs need to be fixed in the source code and potential bugs require a manual postprocessing. Thus it has to be used during the development process instead of an automatic bug treatment during runtime. Furthermore it does not guarantee the completeness of the detection of present bugs. Hence the tool may produce false negatives which can lead to a false sense of security [5].

4.1 Future Work

Although a theoretical prove of correctness for CETS and benchmark results for both CETS and Parfait exist, both tools are not yet publicly available. Thus future work on CETS may include a public release of precompiled LLVM-GCC binaries including CETS for different platforms.

Since Parfait is designed as a framework, releasing precompiled binaries may not be reasonable. Instead future work on Parfait could include pre-assembling layers of analyses for other bug types beside buffer overflows (see section 3).

References


