Enforcing Security in Low Level Languages using Separation Logic

RSMG Meeting III (Thesis Proposal)

2nd March 2009

Horia V. Corcalciuc
University of Birmingham

Contents

1 Types of Exploits and Attacks

An exploit is generally a piece of software containing a sequence of code or commands that take advantage of a flaw in a program in order to cause unintended behavior in computer software or hardware. The most frequent result of such a program is either privilege escalation, in which the attacker gains access to the system, or denial of service in which the system is rendered unusable for its meant purpose. These exploits can usually be classified in two separate divisions: remote and local. Remote exploits focus on gaining access across the network and local exploits usually focus on raising the privilege of the attacker to an administrative level where he can control the machine entirely. In what we have said, we bluntly disregard the network as being just means for data flow and although protocol attacks are generally based on flaws of concept, the other class of exploits focuses on software flaws in machines providing services. Sometimes these two collude and this may be seen frequently in denial in service attacks in which the target machine is, in essence, overwhelmed by the number of access requests to its services.

Having described attacks by the means though which an attacker gains access to a system, we can further classify them statistically by the type or method if exploitation. These are roughly one of the following basic types:

- Overflows
  - Buffer, heap, stack overflows
  - Integer overflows
- Format string attacks
- Code injections
- Race conditions

The divisions are purely informative and describe some basic containers for the large number of exploits designed by attackers some of which are actually combinations of the former. There are other types yet by abstraction we may classify them as a subdivision of one of the above containers.

One big subdivision, as we can observe, tends to be the overflow attacks. It all started with an underground paper, published in Phrack, by a certain Aleph One[33]which explains in detail how a buffer overflow works. Avoiding the crude detail, buffer overflows are mainly due to the fact that some languages, such as C, do not focus on enforcing a precise policy when handling data and memory and this is seen as a task the programmer must accomplish in order to guarantee the security of his piece of software. This is generally preferred since it provides the programmer a lot of power over memory and lets him decide what to do with memory blocks rather than enforcing a certain programming style. The result of that may for example be, the ability to write operating systems but the side effect may also be the ability to write vulnerable software. Overflows are one of the key problems when we discuss software security since they are so easily overlooked without having very precise memory management as a goal. Buffer overflows are considered to be the most popular attack method on the Internet since the release of the Morris worm[34] in 1988. The attack

focuses on sending a sufficiently long string to the insecure application in order to overflow a certain buffer sufficiently close to a return address. The overflow thus forces the execution to point to the buffer and execute some code injected by the attacker.

Consider the following code which illustrates a simple buffer overflow:

```
void f(char *data) {
   ...
   char s[10];

   /* No bounds check done */
   strcat(s, data);
   ...
}

void main(int argc, char *argv)
{
   f(argv[1]);
}
```

This code upon execution would send the contents of the command line parameter to the function f where it will be copied into the memory block pointed to by the character pointer `s`. The pointer `s` points to a memory block of 10 characters. If the command line arguments exceed this length then memory past the 10 character bound will be overwritten. Suppose the figure below is an illustration of the stack before and after the attack. If the attacker manages to find the location of the return address, he may overwrite everything up to the return address and make it point to any arbitrary address in the stack. In this case, he pads the stack with some code which will be executed once he overwrites the function's return address. The injected code can be anything as long as it would fit the storage between the buffer and the return address. In case it is smaller, the rest of the stack up to the return address may be padded by garbage.

If the injected code would, for example, include a simple shell, the attack would grant access to the attacker under the privileges of the running process.

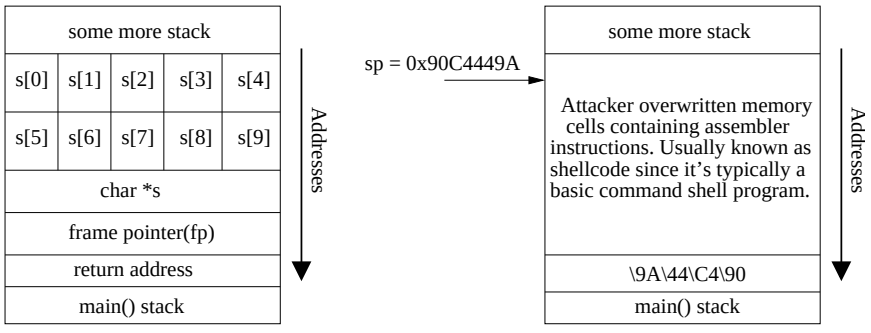


Figure 1: The return address is overwritten so that upon return of the function, the injected code will be executed.

Other problems are also to be seen such as memory corruption cases which the standard C base-compiler language does not pickup. Whether it is GCC or any other ANSI C compiler, heap corruption have been a standard for remote exploits throughout several decades. What really happens in such cases is that memory is being overwritten past the declared bounds of a piece of stack. This leads further to the ability to make concerned programs execute arbitrary code. The problem is not too serious when dealing with local programs in a controlled environment but when one talks about system services, eventually networked system daemons, the problem becomes very serious. This type of attack is primarily due to the fact that there is no bounds checking done on the memory pointed to by `s`. This isn't the case in high-level languages such as Java. They would not accept to overwrite anything in the stack by performing simple bounds checking. This can be viewed as a limitation or a feature depending on the purpose of the program yet it is the primary difference between a "safe language" and an "unsafe" one in case the former prevents the programmer from writing insecure code.

2 Secure Languages and Tools

In order to circumvent these problems, certain dialects of C such as Cyclone [13] or Vault [14] implement mechanisms to implement safety at the language level. These measures usually include things such as bounds checking and regions to prevent access to memory not intended for the program's functionality. There are two types of checks which can be done: run-time checks and static checks. Programming logic, such as Hoare [20] and its extension separation logic [36], however, provide some means of modeling the safety of a program using pre- and postconditions. If we take the above example into a consideration we can model some sort of bounds checking to prevent the programming flaw[11].

Looking back at Vault, we can see that it provides parts of this logic by implementing definable states. The Vault project at Microsoft Research [14] focuses on controlling memory and life-span of user-defined variables or states. Similar to C, Vault contains function prototypes but introduces a "tracked" keyword to identify resources to be manipulated with care. A function can thus create resources, dispose of resources and change the symbolic states associated with these tracked resources. The following function prototypes are an example of how this is accomplished:

```
tracked($F)FILE open_read(string)[new $F @ readable ];
void write(tracked($F)FILE,char)[$F @ writable ];
```

In the example above `$` is the symbol for key. Vault uses keys to reference resources. Each resource has an associated key which, at its turn, can be in a certain state. As we can observe, this type system is very close to traditional Hoare[21] logic. It is based on pre- and postconditions; that is, a program may find itself in a state before executing a block of code and in another state after the piece of code was executed. This is very useful in order to eliminate all possible confusion arising from the weak checks done by the compiler. The following table depicts the syntax of such a construct:

Δ	Precondition	Postcondition
`$K@s`	`$K@s`	`$K@s`
`$K@s→@t`	`$K@s`	`$K@t`
`-$K@s`	`$K@s`	`not $K`
`+$K@s`	`not $K`	`$K@s`
`new $K@s`		`$K@s`

where Δ is the shorthand combining the pre- and postcondition in a single construct. The table is interpreted as follows:

1. The key `k` is in a state `s` before executing a block of code. After execution, the key `k` finds itself in the same state `s`.
2. The key `k` is in a state `s` before execution and finds itself in a state `t` after.
3. The key `k` exists before executing a block of code but is disposed after the code is executed.
4. The key `k` does not exist before execution but is created during execution and finds itself in state `s` at the end.
5. The key `k` does not initially exist and is created with the default state `s`.

A feature similar to what Cyclone already has implemented consists in "region handling". Basically a region[17] is a part of the heap. Objects are allocated from that heap and the whole region is disposed of entirely after use making further calls to that certain region impossible. Similar to Cyclone, this feature would prevent a vast amount of errors common to programming and would hinder buffer overflows entirely.

The Microsoft Research web-site gives a lot of pointers on how to use Vault and most importantly we can see that the code is publicly available. Developers at Microsoft have gone on and started a new project called Singularity [22]. This new system shares Vault concepts but expands it further using a new language Sing# which should be an offspring of C#. Regrettably, the available articles focus on the Singularity Operating System rather than on Sing# and the most quoted article is just an informal description of this new secure operating system. Furthermore, there are very few articles on Sing# although Microsoft Research claims that they have "reached Singularity v1.0" [23]. The SDK they mention which is available only to some parties most probably includes the compiler in its binary form but no documentation on the syntax thus making it practically impossible to determine if and how much of Vault is used[16]. The effort will be to use Cyclone as a base compiler since it is released as Open Source and build up the features of Vault trying to fix and implement new rules to bring it as close as possible to Hoare and separation logic.

It is important to mention that the current research field does not currently have a state of the art compiler implementing the features we are after. There are a few vague papers and a tool called "Smallfoot" [3] which describes a method to verify already existent programs against logical composition rules. However, the code only finds errors or potential pitfalls. It does not suggest

corrections and it does not implement a language by itself. Further research include static checkers for code pertaining to different domains (drivers, etc...) yet none of them solve the problem on an elementary level. What we are after is some way to circumvent all errors by implementing a type safe language which will automatically exclude errors related to concurrency and memory corruption. This is hardly done since current state of the art tools only have the ability to check and verify code instead of offering a possibility to "create" secure code. We can take a look at Valgrind[38], which is a well known tool to check for memory leaks at run-time or we can look at programs such as Smallfoot yet neither of them offer any insight on the proposed research. They do implement static rule checking through parsers or emulating a VM to check the stack but they do not provide means to correct the faulty code or implement methods which Vault and Cyclone offered before being abandoned.

2.1 Vault

Vault is considered to be a safe version of the C programming language which is developed at Microsoft Research. It provides the same level of safety as languages like C# but it allows a programmer to retain control over data layout and lifetime. Unlike Cyclone, which targets certain features which implications to security, Vault goes another way having code safety in mind and modifying a wide range of features. For example, most of the modifications are at a higher level concerning functions in general or constructs, variants and aggregate types and also by bringing in some concepts of object oriented programming languages like modules and generics. Vault has to be seen as an attempt to modify the language as a whole rather than implementing constructs to avoid certain common problems. Somehow, given Vault constructs the dialect generally focuses on data access control rather than security concerns. Like we noticed previously, Vault implements constructs which are based mostly on access control by allowing the programmer to specify when and under what circumstance a certain variable may be accessed. Along the way it may seems redundant to let the programmer implement these checks since it does require the programmer to use constructs in certain places where there "might" be an issue but that may be unknown.

Vault implements modules and interfaces which are the main highlights of this dialect. Modules are a collection of type, variable and function definitions. One can declare them to be inner or outer modules just like structs and they resemble a simple class object. Encapsulation can be done by using the static keyword when declaring variables or functions inside the module. Interfaces provide, in turn, encapsulation and information hiding similar to a C header file. Similar to Java, an interface acts as a contract between a module implementation an a module client. A module can also claim or adopt an interface and implement the details.

Arguably, because of these additions Vault tends to resemble Java more than C by implementing objects, complex scoping and all the main characteristics one would find in an object oriented programming language.

2.2 Cyclone

Cyclone is a safe dialect of the C programming language designed specifically to avoid buffer overflows and similar problems in C programs. Cyclone stated as a joint project of AT&T Labs Research and Greg Morrisett's group at Cornell in 2001 and the first version was released on 8th of May 2006. However since 2006 there are no major updates to the project and development has halted. Using a couple of restrictions, Cyclone claims to maintain the look and performance of C programs:

- `NULL` checks are inserted to prevent segmentation faults for normal `*`-type pointers.
- Pointer arithmetic is restricted and pointers are split up into various other subtypes (those supporting pointer arithmetic and bounds checking like fat pointers, those which are never `NULL` and don't require `NULL` checks like `?`-pointers and normal `*`-type pointers).
- Pointers must be initialized before use.
- Cyclone implements regions to deal with dangling pointers.
- Only safe casts and unions are allowed (transitions from a certain pointer type to another are restricted).
- `goto` is disallowed into scopes.
- `switch` labels in different scopes are disallowed.
- Functions returning pointers must execute return.
- `setjmp`, `longjmp` and more generally concurrency are not supported (however Cyclone allows thread libraries and agrees to compile yet the resulting program does not create any threads).
- arrays and strings are automatically converted to `?`-pointers.

Cyclone also implements a few extensions to the C programming language bringing, like Vault, some elements of object programming languages:

- Garbage collection for heap-allocated values.
- Exceptions
- Polymorphism which replaces some uses of `void *`

The main strength of Cyclone comes from splitting pointers into three categories. We can show a case study of code in traditional C and its safe counterpart in the Cyclone dialect:

```
/* Standard C Variant */
int strlen(const char *s) {
   int i = 0;
   if (!s) return 0;
   /* UNSAFE */
   while(*s) i++;
   return i;
}
```

```
/* Cyclone variant */
int strlen(const char ?s) {
   int i, n;
   if (!s) return 0;
   n = s.size;
   for(i=0; i<n; i++, s++)
      if(!*s) return i;
   return n;
}
```

As we can see in the traditional C variant, the loop would be unsafe if `s` wouldn't be `NULL` terminated. The Cyclone variant forcibly converts the pointer to a `NULL` terminated pointer and also allows to check the size of `s` using a reference construct. In the traditional C example, if `s` wouldn't be `NULL` terminated the result would be undefined behavior (typically, although not necessarily, a `SIGSEGV` signal being sent to the application).

Taking a simpler example, suppose the following function was written in C:

```
int fun(int *);
```

if the function `fun` wouldn't include a null check for the pointer argument, it would again produce undefined behavior. With Cyclone, it would be just a matter of rewriting the code:

```
int fun(int @);
```

This is telling the compiler that the argument of the function `fun` should never be `NULL`. Of course, it is important to emphasize the fact that this problem could also be avoided in standard C by just including a `NULL` pointer check inside the function which would return (like in the previous example) if the pointer would be found to be `NULL`.

Overall, Cyclone bring in some new features to the C programming language also adding some concepts of object programming language. It is important to state that these features are not based on any type of logic and more on intuitive reasoning about issues governing code writing. In other words, the set of additions try to circumvent common mistakes made by programmers yet they could be avoided altogether by a careful programmer implementing his own checks or managing memory properly.

2.3 CCured

CCured[41] is a source parser which takes plain C code and modifies it by inserting run-time checks in order to prevent memory misuse. When compiled, the program is considered to prevent all memory violations and stop rather than corrupting memory. This of course comes with a performance impact yet this can be improved by manually adding checks or by having CCured skip parts of code that may be to difficult to parse. CCured introduces five primary new pointer types. Amongst the main kinds we find `SAFE` (not used for pointer arithmetic and requires just `NULL`-checks before dereference), `SEQ` (used in pointer arithmetic but no unsafe casts - it requires a bounds-check and a `NULL`-check), `WILD` (used in pointer arithmetic, unsafe casts and does

bounds, `NULL` and dynamic type checks) and some minor ones like `RTTI`, `FSWQ`. At compile time CCured suggests ways to make inferred `WILD` pointers into `SAFE` and `SEQ` pointers. Both `SEQ` and `WILD` pointers are fat-type Cyclone pointers and there are external function wrappers to convert from fat to normal pointers[29]. The difference to Cyclone is that it infers pointer kinds based on use rather than declaration and each pointer reflects how safe it is to use it. There are also some minor unpleasantries which arise from these pointer types. A common problem, also discussed in the Cyclone article is related to vararg functions which require changes. `sizeof()`, takes for example a variable and will not accept a type.

However, CCured doesn't always produce the desired output and although the developers claim that they have successfully ported many opensource projects such as sendmail, bind and even Apache modules, CCured requires a lot of user intervention. Sadly enough, very few of the attacks mentioned above in the above list are solved. CCured's only solves problem related to pointer types and types that relate to pointers thus ignoring a lot of the problems mentioned before. For example, concurrency is far away from CCured's analysis. Like Cyclone, CCured works by substituting memory related functions (like `malloc()`, `calloc()`, `free()`, etc...) yet, of course, this may cause compatibility issues. CCured also implements a garbage collector which comes more as an addition than a fix. Given the scope of this thesis, CCured is relevant in a functional way: the idea of transforming or translating code to a secure version is one desirable feature[30]. However, we would rather replace the run-time checks by static checks based on different types of logic (as we will discuss in the following chapters) and most certainly the garbage collector is a redundant feature we would not like to implement. Also, upon error, CCured aborts with a rather vague message so that one is compelled to use debugging tools to find the exact location of the error. This, of course, for complex bugs is totally unfeasible.

2.4 CRED

CRED[37] stands for C Range Error Detector and generally focuses on avoiding buffer overflows. It does so by trying to replace every out-of-bounds pointer value with the address of a special object created for that value. Before dereferencing, any pointer derived from that address is bounds checked. It doesn't modify the pointer representation however and thus works better with uninstrumented code. By creating an object table it records the base address and the size of all memory objects either static, from the heap or stack and stores them there. At dereference, it looks up the pointer in the object table and performs bounds checks. However, pointer arithmetic has several problems and we will discuss it since it is similar to what we do later on in a case-study example.

For in-bounds pointer arithmetic, the address computed from an in-bounds pointer must refer to the same object as that pointer. Then a check is done to see if the pointer is in-bounds. If so, it finds its referent and the final result of arithmetic must fall within the referent's bounds.

Four out-of-bounds pointer arithmetic, the address computed from an out-of-bounds pointer must refer to the same object as that pointer's referent. Then, for each out-of-bounds pointer CRED creates an out-of-bounds object in the heap, the out-of-bounds pointer points to that out-of-bounds object

which contains the original out-of-bounds pointer address and referent.

After each address computation, it checks if the result is out-of-bounds and if so it creates an out-of-bound object and stores it in the out-of-bounds table. If the pointer is not used in in-bounds pointer arithmetic it performs and out-of-bounds table lookup. And finally, if an object is deallocated it removes it from the out-of-bounds table.

CRED only checks strings and even though string buffers are most commonly overflowed it doesn't offer a complete solutions and detect non-string attacks and also doesn't analyze type casts.

2.5 GCC Protections

GCC itself offers some minor protections included in the compiler itself and triggered by specific options. We would like to enumerate some of these options present in the most recent compilers since they employ techniques which have been used before (ie: honeypots) not only in programming but also in operating system design and considered safe and stable enough to be included in the mainstream distribution of gcc. These security features are both run-time as well as static depending on the type of protection(s) the user selects by passing a compile-time option to gcc. A lot of extensive work is being done on the trunk of the gcc compiler but not all features are considered to useful and some of them may be circumvented quite easily[7]. It is interesting to note that protections are also offered from other domains, even up to the processor hardware level where, for example, we have the "NX" (No-eXecute) processor flag implemented in newer processors which allows the system to control which part of memory is to be executed. Data memory may be flagged as being non-executable and a program memory may be flagged as non-writeable which can prevent certain types of buffer overflow exploits from working since it wouldn't allow memory overwrites or the processor simply wouldn't execute that set of instructions. Another, rather controversial, feature is the implementation of PIE executables which allows the programmer to make the executable load at a different address each time it starts and randomizes addresses. This is rather controversial since it is basically security through obscurity and it is a known fact that methods have been devised to circumvent and predict the memory addresses. However, to the inexperienced attacker it does pose a problem. This type of feature is very similar to one of the functions of the grsec kernel extensions which randomizes PIDs every time a process runs (which eventually leads to severe problems and compatibility issues with programs relying on the process table architecture) However, these features are tied very close to the operating system and we shall focus on two of these features covered by the latest gcc compiler:

- Source fortification (triggered by specifying `-DFORTIFY_SOURCE` to the gcc compiler). This is a static check, implementing compile time buffer checks and is meant to prevent buffer overflows in C programs. It's relies on a very simple idea that in certain cases we actually know the length of a buffer (such as statically allocated buffers where the size of the buffer on the stack is fixed or if the buffer was just allocated using `malloc()`). If we know the size, it is fairly easy to make sure that functions which operate

on that buffer will not overflow. This is a relatively new feature (starting gcc 4.2 and up) and has also been the highlight for certain claims that it is fairly easy to fool these checks and circumvent it all together.

- SSP[39] (Smash Stack Protection) is another feature which can be triggered by the `-fstack-protector` flag on recent gcc compiler versions. It starts off with the old concept of honeypots: certain "honeypots" are placed on the stack, allegorically to attract the attackers symbolized by the "bees" and trigger a certain logic. Technically, a certain value (the allegorical honeypot) is placed on the stack and once the attacker tries to overwrite the memory, the checks introduced by the pre-processor will notice that a violation of memory has occurred and that part of memory may have been corrupted. SSP moves further on to variable reordering to prevent corruption of pointers by reordering local variables to place buffers after pointers and copies pointers in function arguments to an area preceding local variable buffers in order to avoid the corruption of arbitrary memory locations[10].

Additionally, Linux benefits from a dynamic library called libsafe/libverify[5] available from Avaya Labs Research which, once linked to a binary, intercepts common C libary function calls which are susceptible to buffer overlows and performs bounds checks making sure that frame pointers and return addresses can not be overwritten. It is in some ways similar to SSP mentioned previously, however the greatest benefit is that it does not require a recompilation of the program itself and any binary can link against it. This transparency is remarkable and is one of the desired goals of this project.

2.6 Splint

Splint[26] was formerly named LCLint and is a tool for statically checking C programs and is derived from the historical tool called "lint"[24] now being part of a broader project called LARCH[19]. It can be used in two different modes: by inserting source code with information related to usage or by writing specifications in separate files using LCL(LARCH interface Language for C). Splint addresses several problems, out of which we mention:

- NULL pointer dereferencing
- Use of unallocated memory
- Type mismatch
- Bad aliasing
- Infinite loops
- Buffer overflows
- Badly implemented macros

Take for instance a typical NULL pointer dereferencing example with it is Splint variant:

```
char f(char *ptr)
{
   return *ptr;
}
```

```
char f(/*@null@*/ char *ptr)
{
   return *ptr;
}
```

The notation `/*@null@*/` tells Splint that the pointer might be `NULL`. Splint would in this case report and error saying that there may be a dereferencing of a `NULL` pointer. Of course, a possible fix for this would be:

```
char f(/*@null@*/ char *ptr)
{
   if(ptr == NULL) return "\0";
   return *ptr;
}
```

Which would compile without errors using Splint. Another practical example is the use of parameter "getters" and "setters", like the ones we meet in C#. Splint allows us to define what a certain parameter passed to a function represents. Consider the following code and the Splint annotated counterpart:

```
extern void set(int *x);
extern void get(int *x);
extern int huh(int *x);
...
int f(int *x, int i)
{
   switch(i)
   {
      case 1:
         return *x;
         break;
      case 2:
         return get(x);
         break;
      case 3:
         return set(x);
         break;
      default:
         return huh(x);
         break;
   }
}
```

```
extern void set(/*@out@*/ int *x);
extern void get(/*@in@*/ int *x);
extern int huh(int *x);
...
int f(/*@out@*/ int *x, int i)
{
   switch(i)
   {
      case 1:
         return *x;
         break;
      case 2:
         return get(x);
         break;
      case 3:
         return set(x);
         break;
      default:
         return huh(x);
         break;
   }
}
```

This annotation specifies the role of the function parameters since the functions may be declared externally and we won't be able to logically analyze if the function calls are correct. Thus splint allows the programmer to specify the specific role of every parameter by either being and input parameter `/*@in@*/` or an output parameter `/*@out@*/`. In the previous example, Splint would

report a misuse on storage `x` not being correctly defined for `get(x)` and for `huh(x)`.

Splint also adds alternative functions in order to accomplish tasks which can be seen as unsafe, for example say, the comparison of two boolean values, which are all included in Splint's library. These specifications, as we have seen in the previous examples, allow us to reason about code and pay close attention to what we generally do in a program. Such extra constructs are considered to be static-checks and can be analyzed by Splint offline without any overhead to the run-time environment. This is a key point we have to highlight since, as we shall see further on, this is the desired operation for our proposed work. Previous examples of Vault and Cyclone include a vast ammount of run-time checks (the top of those being Cyclone with an additional garbage-collector) which increase the work-load considerably. Also, the good thing about static-checks is that it allows us to reason with logic about the behavior of a program but, in the case of Splint, requires extra help from the programmer to annotate things that may not be known until it is too late. Looking at the first example with dereferencing `NULL` pointers it is clear that the programmer must know while writing the `f` function specification that the character pointer `ptr` might be `NULL`. The question which arises from this is that it is unlikely that the programmer is conscious that the character pointer `ptr` might be `NULL` since if the programmer would know that, then it would be likely that the programmer would have written a `NULL` pointer check in the first place. It is imperative that the work we propose would spot these inconsistencies by itself through logic reasoning rather than user-specified annotations in order to overcome this paradox.

2.7 CQUAL

CQUAL[18] is a type-based analysis tool for finding bugs in C programs. It extends the language by allowing user-defined type qualifiers which are used in the same way as the standard C type qualifiers, like const for example. It can note that values are "tainted" or "untainted" (similar to Perl's -T taint checking in which Perl treats all user supplied input as being malicious unless the programmer explicitly confirms the validity of the provided data). The programmer annotates the program in a few places and CQUAL performs qualifier inferences to check whether the annotations are correct.

To see how this works, suppose we define a type quantifier "unchecked" which we use to mark an object that has not been authorized.

```
struct file * $unchecked fp;
```

The previous declaration states that the file object `fp` has not been checked. This could for example be useful if a certain function expects to find an argument to be checked before it could operate on it. CQUAL performs checks to see if there is a type violation. The following code segment depicts a type violation:

```
void f(struct file *$checked fp);
void g(void)
{
```

```
    struct file * $unchecked fp;
    ...
    f(fp);
    ...
}
```

The function `f` expects a `checked` file pointer as parameter but the passed pointer is of type `unchecked`. This works since CQUAL also introduces the notion of "lattices" which, for our example would be something like:

```
partial order {
    $checked < $unchecked
}
```

which is basically saying that `$checked` is a subtype of `$unchecked`. A lattice is a partially ordered set in which all nonempty finite subset have a least upper bound and a greatest lower bound. In our snippet above the lower bound would be considered to to `$checked` and the upper bound would be `$unchecked`. This also has the property that a `$checked` type can be used wherever an `$unchecked` type is expected but the reverse of that would result in a type violation error.

CQUAL is pretty simple and from what we have seen could be used, for example, to detect format-string vulnerabilities in C programs. A derivation of CQUAL has also been used to find Y2K bugs in C programs. CQUAL has also been used to test[44] the implementation of LSM modules[42]. CQUAL doesn't claim to solve all the problems or security or code safety issues pertaining to a C program but it introduces an useful concept which can be used to prevent several types of attacks as we have seen in the case of format string attacks. We present CQUAL since the idea of implementing a construct to solve a certain problem is appealing as this shall be discussed further on in the "Conclusions and Proposed Work" chapter of this thesis proposal. CQUAL is licensed under GPL and publicly available.

3 Flavors of Logic

3.1 Propositional Logic

Propositional logic, also known as sentenial logic or statement logic, is the branch of logic that studies ways of joining or modifying entire propositions, statements or sentences to form other propositions, statements or sentences as well as logical relationships and properties that are derived from these methods of combining or altering statements. The simplest statements are considered atomic units and hence propositional logic does not deal with the logical properties and relations that depend upon parts of statements which are not themselves statements on their own such as the subject and predicate of a statement. The most thoroughly researched branch of propositional logic is the classical truth-functional propositional logic which studies logical operators and connectives that produce complex statements depending on the truth values of the simpler statements used to build them up. This makes propositional logic to be a formal system for performing and studying logical reasoning and as such it has a precisely defined grammar. Thus we can write the well formed formula:

`<Wff> ::= <Wff>` → `<Disjunction> | <Disjunction>`

`<Disjunction> ::= <Disjunction> or <Conjunction> | <Conjunction>`

`<Conjunction> ::= <Conjunction> and <Literal> | <Literal>`

`<Literal> ::= not <Literal> | <Variable> | (<Wff>)`

adding "`&`", "`.`" or "∧" for "`and`", "`~`" or "¬" for "`not`", and "`+`" or "∨" for "`or`".

which identify the formal language. Proofs as the result of the propositional calculus, are sequences of WWFs with certain properties; the final WWF in the sequence is called a theorem when it plays a significant tole in the theory or a lemma in case it plays a secondary role in proving the theorem.

Propositional calculus is a formal system in which formulas representing propositions can be formed by combining atomic propositions using logical connectives and a system of formal proof rules [28]. These rules allow us to derive other formula based on a set of formula that are assumed to be true. In turn, these would allow us to build a proof for a certain assumption by either pushing the axioms forward using the rules or symmetrically by tracing backward.

A set of rules for propositional logic can consist in the following:

- Reductio ad absurdum

This is generally done by assuming the contrary of what there is to prove and pushing forward until a contradiction is reached. In which case, the only valid possibility is that our assumption is correct.

- Double negative elimination

For example: ¬¬`A` ⇒ `A`

- Conjunction introduction

For example: $\mathtt{A}, \mathtt{B} \Rightarrow (\mathtt{A} \wedge \mathtt{B})$

- Conjunction elimination

For example: $(\mathtt{A} \wedge \mathtt{B}) \Rightarrow \mathtt{A}$ and $(\mathtt{A} \wedge \mathtt{B}) \Rightarrow \mathtt{B}$

- Disjunction introduction

For example: $\mathtt{A} \Rightarrow (\mathtt{A} \vee \mathtt{B})$ and $\mathtt{B} \Rightarrow (\mathtt{A} \vee \mathtt{B})$

- Disjunction elimination

For example: $(\mathtt{A} \vee \mathtt{B}), (\mathtt{A} \rightarrow \mathtt{C}), (\mathtt{B} \rightarrow \mathtt{C}) \Rightarrow \mathtt{C}$

- Biconditional introduction

For example: $(\mathtt{A} \rightarrow \mathtt{B}), (\mathtt{B} \rightarrow \mathtt{A}) \Rightarrow (\mathtt{A} \leftrightarrow \mathtt{B})$

- Biconditional elimination

For example: $(\mathtt{A} \leftrightarrow \mathtt{B}) \Rightarrow (\mathtt{A} \rightarrow \mathtt{B})$ and $(\mathtt{A} \leftrightarrow \mathtt{B}) \Rightarrow (\mathtt{B} \rightarrow \mathtt{A})$

- Modus ponens

For example: $\mathtt{A}, (\mathtt{A} \rightarrow \mathtt{B}) \Rightarrow \mathtt{B}$

and lastly, by assuming a certain unproven hypothesis in the premise to be "temporarily" true in order to see if we can infer a certain formula, we have:

- Conditional proofs

A proof is always necessary in order to validate any statement we make. We use logical reasoning and by breaking down a statement with rules we reduce the problem to a series of hypothesises and axioms. Obviously, the reverse is also possible: by using axioms, hypothesises and rules we are able to deduce our statement. In doing so we prove that a statement holds under the axioms and our hypothesises.

Given the following two definitions of introduction and elimination:

$$\text{Introduction:} \frac{\Gamma, \mathtt{A} \vdash \mathtt{B}}{\Gamma \vdash \mathtt{A} \rightarrow \mathtt{B}} \quad \text{Elimination:} \frac{\Gamma \vdash \mathtt{A} \rightarrow \mathtt{B} \quad \Gamma \vdash \mathtt{A}}{\Gamma \vdash \mathtt{B}}$$

Where $\Gamma$is said to be the context. Taking these rules, we can use them to construct proofs for various statements.

For example, take a simple conditional case. We want to prove that:

$(\mathtt{A} \rightarrow (\mathtt{B} \rightarrow \mathtt{C})) \rightarrow ((\mathtt{A} \rightarrow \mathtt{B}) \rightarrow (\mathtt{A} \rightarrow \mathtt{C}))$

Using the same rules, we can write:

1. $\mathtt{A} \rightarrow \mathtt{B}$ [hypothesis]
2. $\mathtt{B} \rightarrow \mathtt{C}$ [hypothesis]
3. $\mathtt{A}$ [hypothesis]
4. $\mathtt{B}$ [Elimination using 1 and 3]

5. C [Elimination using 2 and 4]

6. A → C [Introduction using 3 to 5]

7. (A → B) → (A → C) [Introduction using 1 and 6]

8. A → (B → C) [Introduction using 3 and 2]

9. (A → (B → C)) → ((A → B) → (A → C)) [Introduction using 8 and 7]

Take another example:

(A → B) → ((B → C) → (A → C))

In order to prove it, we use Introduction and Elimination via a series of deductive steps:

1. (A → B) [hypothesis]

2. (B → C) [hypothesis]

3. A[hypothesis]

4. B [Elimination using 1. and 3.]

5. C [Elimination using 2. and 4.]

6. (A → C) [Introduction using 3. to 5.]

7. (B → C) → (A → C) [Introduction using 2. and 6.]

8. (A → B) → ((B → C) → (A → C)) [Introduction using 1. and 7.]

We can then construct the corresponding proof trees using the axioms and the introduction and elimination rules as depicted in Figure 2.

$$\frac{}{\underbrace{A_1,\ \ldots,\ A_n}_{\Gamma} \vdash A_i}\ (Axiom) \qquad \underbrace{\frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}\ (\to Elim)}_{Modus\ Ponens} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ (\to Intro)$$

Proof for $(A \to (B \to C)) \to ((A \to B) \to (A \to C))$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \to B \vdash A \to B}(Ax) \quad \dfrac{}{B \vdash B}}{A \to B, A \vdash B}(\to E) \quad \dfrac{\dfrac{}{B \to C \vdash B \to C}(Ax) \quad \dfrac{}{C \vdash C}}{B \to C, B \vdash C}(\to E) \quad \dfrac{}{A \vdash A}(Ax)}{A, B \to C, A \to B, A \vdash C}(\to I)}{A \vdash B \to C, A \to B \vdash A \to C}(\to I)}{A \to (B \to C) \vdash (A \to B) \to (A \to C)}(\to I)}{\vdash (A \to (B \to C)) \to ((A \to B) \to (A \to C))}(\to I)$$

Proof for $(A \to B) \to ((B \to C) \to (A \to C))$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \to B \vdash A \to B}(Ax) \quad \dfrac{}{A \vdash A}(Ax)}{A \to B, A \vdash B}(\to E) \quad \dfrac{\dfrac{}{B \to C \vdash B \to C}(Ax) \quad \dfrac{}{B \vdash B}(Ax)}{B \to C, B \vdash C}(\to E) \quad \dfrac{}{A \vdash A}(Ax)}{A \to B, B \to C, A \vdash C}(\to I)}{A \to B, B \to C \vdash (A \to C)}(\to I)}{A \to B \vdash (B \to C) \to (A \to C)}(\to I)}{\vdash (A \to B) \to ((B \to C) \to (A \to C))}(\to I)$$

Figure 2: Intuitionistic proof

3.2 Linear Logic

Linear logic is a type of substructural logic (we take this privilege to note that separation logic is also a type of substructural logic) which denies the rules of weakening and contradiction. The concept of "hypotheses as resources" is the base of linear logic and it basically means that each hypothesis is consumed exactly once in a proof. Once a hypothesis is consumed, it can not be used again. This differs from the previous logic we had where the judgment is based on truth, which may be used as many times as necessary.

Linear logic introduces a few new logical connectives. The most important being the "linear implication" $\mathtt{A} \multimap \mathtt{B}$ which basically says that an action A is a cause of an action B. A formula A can be regarded as a resource which is consumed by the linear implication. It is the most important feature of linear logic and also of the programming based on it. In classical logic the truth value of the formula A after the implication $\mathtt{A} \rightarrow \mathtt{B}$ remains that same meaning that the same rule can be applied over and over again as many times as it is necessary. Classical implication can be ported using a modal operator and by writing $(!\mathtt{A}) \multimap \mathtt{B}$ meaning that we can use resource A repeatedly. Linear logic also defines the "multiplicative conjunction" represented by $(\mathtt{A} \otimes \mathtt{B})$ and saying that both actions A and B will be done. There is also the additive conjunction represented by (A&B)expressing that only one of the actions A or B will be performed at choice. Another one is the "additive disjunction" represented by $(\mathtt{A} \oplus \mathtt{B})$ which also says that only one of the actions A or B will be performed yet it is unknown which one of them. Finally, there's "multiplicative disjunction" $(\mathtt{A} \varrho \mathtt{B})$which is saying that if A is not performed then B is done, or vice versa: if B is not performed then A is performed.

In order to illustrate this [27], we can make a very rough comparison. Suppose that we have milk and we use it to make icecream. Which is very desirable since it describes a real world situation; if we transform the milk into icecream the milk got consumed and can not obviously be used again. Consider the following reasoning schemata:

1. "we have milk" [hypothesis]
2. "we have milk $\Rightarrow$we have icecream" [hypothesis]
3. "we have icecream" [Modus ponens from 1. and 2.]
4. "we have milk $\wedge$ we have icecream" [Conjunction using 1. and 3.]

Which is problematic since at the end we seem to have both milk and icecream. This is the very nature of linear logic, whether it is about icecream or resources in general, the quantity of resources is not a free fact to be used and disposed of at will, like truth, it must be accounted for in every state.

Using linear logic, we can rephrase the above to: "A kilogram of milk is transformed into half a kilogram of icecream". And then re-write the schemata using linear logic this time:

1. "we have milk" [hypothesis]
2. "we have milk $\multimap$we have icecream" [hypothesis]
3. "we have icecream" [Modus ponens 1. and 2.]

And this time we have the milk consumed without being able to mention it again eliminating step 4. in the previous schemata and making our proof sound (actually there is one more step but we don't mention the short version of the schemata for brevity purposes).

Again, like we did previously, we can take our examples and construct the linear logic proof tree version as shown in Figure 3. Note that this is not applicable to our first example since A would already be consumed and we won't be able to reuse it again as in the classical logic version. However, the !(bang) operator allows us to hard-wire the classical logic version of $\mathtt{A} \rightarrow \mathtt{B}$. Writing $!\mathtt{A} \multimap \mathtt{B}$ (which is written in this form for the sake of example) is basically saying that A would be consumed but we have an endless supply of A's [35]. We also use a new rule called dereliction in order to select on A out of our supply of A's. Basically this is saying that if we have an endless supply of A's, we select one out of them. The problem is that we can not apply $\multimap \mathtt{B}$ to an $!\mathtt{A}$; we can only apply it to one single A and hence the need for Dereliction. The axioms are also modified to reflect the change brought by the resource-consumption concept. Like the multiplicative conjunction, the elimination rule uses separate contexts which are illustrated in our example by Δ and Γ. This bears a close similarity, as we shall see further on, to separation logic's $*$ which splits the heap and allows us to reason in a different context.

3.2.1 Linear Logic in Computer Science

Logic, more precisely proof-theory, is focused on formal proof systems as we have seen: intuitionistic predicate calculus, classical predicate calculus, arithmetics, high order calculi and similar hybrids of consistent and structured sets of rules. Intuitionistic logic focuses on the interpretation of reading a statement like $A \Rightarrow B$ as 'if A is given, then I will give B' which differs in nature from the classical approach 'B is true whenever A is true'.

Computer science relies on computational mechanisms such as function application, exception handling, method invocations, variable assignments and similar. It is thus necessary to make the mechanisms of these processes explicit by using a formal language. An event such as $\mathtt{A} \rightarrow \mathtt{B}$ generally describes a transformation from A into B. This is also extensively applicable to automated processes, more precisely Petri nets, where a clear protocol must be implemented by following a series of transformations. One key application of this resource management aspect of linear logic was the development of a functional programming language which replaces garbage collection by explicit duplication [25] operations. Other applications include analyzing the control structure of programs, generalized logic programming and natural language processing. One interesting feature is the ability to give a natural aspect of polynomial time computations in a bounded version of linear logic. This is done, again, by limiting the reuse of specified bounds.

$$\underbrace{\frac{}{\underbrace{A_1,\ \ldots,\ A_n}_{\Gamma} \vdash A_i}}\ (Axiom) \qquad \underbrace{\frac{\Gamma \vdash A \multimap B \quad \Delta \vdash A}{\Gamma, \Delta \vdash B}\ (\multimap Elim)}_{Modus\ Ponens} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}\ (\multimap Intro) \qquad \frac{\Gamma, \vdash !A}{\Gamma \vdash A}(Dereliction)$$

Proof for $(A \multimap (B \multimap C)) \multimap ((!A \multimap B) \to (A \multimap C))$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(A \multimap (B \multimap C)) \vdash (A \multimap (B \multimap C))}(Ax) \quad \dfrac{\dfrac{}{!A \vdash A}(Ax)}{!A \vdash A}(!Der)}{A \multimap (B \multimap C), !A \vdash B \multimap C}(\multimap E) \quad \dfrac{\dfrac{}{A \multimap B \vdash A \multimap B}(Axiom) \quad \dfrac{\dfrac{}{!A \vdash A}(Ax)}{!A \vdash A}(!Der)}{A \multimap B, !A \vdash B}(!Der)}{A \multimap (B \multimap C), A \multimap B, !A !A \vdash C}(\multimap I)}{A \multimap (B \multimap C), A \multimap B, !A \vdash C}(!Con)}{A \multimap (B \multimap C), (A \multimap B) \vdash (!A \multimap C)}(\multimap I)}{A \multimap (B \multimap C) \vdash (A \multimap B) \multimap (!A \multimap C)}(\multimap I)}{\vdash (A \multimap (B \multimap C)) \multimap ((A \multimap B) \multimap (A \multimap C))}(\multimap I)$$

Proof for $(A \multimap B) \multimap ((B \multimap C) \multimap (A \multimap C))$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{A \multimap B \vdash A \multimap B}(Ax) \quad \overline{A \vdash A}(Ax)}{A \multimap B, A \vdash B}(\multimap E) \quad \dfrac{}{B \multimap C \vdash B \multimap C}(Ax)}{A \multimap B, B \multimap C, A \vdash C}(\multimap E)}{A \multimap B, B \multimap C \vdash (A \multimap C)}(\multimap I)}{A \multimap B \vdash (B \multimap C) \multimap (A \multimap C)}(\multimap I)}{\vdash (A \multimap B) \multimap ((B \multimap C) \multimap (A \multimap C))}(\multimap I)$$

Figure 3: Linear logic proof

3.2.2 Vault

Coming back to Vault, we have to note that Vault brings some personal contributions based on linear and intuitionistic logic. This is based on the concept that a certain object or variable in a programming language may have a specific lifetime and it is easier (for example, in cases where we must free up a resource) to reason about them if we combine the two logics together. DILL [1] comes as an answer to that and explores the compatibility and transitions from one logic to the other.

As a repetition, in intuitionistic logic we write constructs such as `A` $\rightarrow$ `B` which can be seen as a function application: "given an `A` we can produce a `B`". This, of course, can be seen as a function which can be applied as many times as necessary, taking a parameter and producing a result. In linear logic, we write constructs such as `A` $\multimap$ `B` which can be seen as a one-way function application: "given an `A` we can produce exactly one `B` by consuming the `A`". Such a function would resemble for example a `free()`, in which we take a parameter and, in this case, delete it after which that parameter can not be accessed again. By using linear logic one can efficiently describe such functions yet it is not sufficient and we need inuitionistic logic for other functions which do not consume parameters since it is obvious that by building a language solely on linear logic would produce unwanted results (every variable, function or object will need to be recreated every time and will be consumed every time after an operation). DILL has sought to combine the two logics together by exploring how far one can combine inuitionistic logic with linear logic in order to acquire the functionality of both. This however leads to very complex constructs and ends up creating the need for another logic (which we describe in the following chapter).

Vault solves the incompatibility between intuitionistic and linear logic by creating two new concepts: adoption and focus[15]. In Vault, all object die and are born as linear types. Non-linear objects are temporarily cast to linear by using the `let!` construct. The following Figure 4 is a representation of the lifespan of an object in Vault depicting the transition from linear to non-linear and vice versa through the means of adoption and focus. By using adoption and focus it is possible to change the type of an object however, as seen in the figure, all objects are born and die with a linear type:

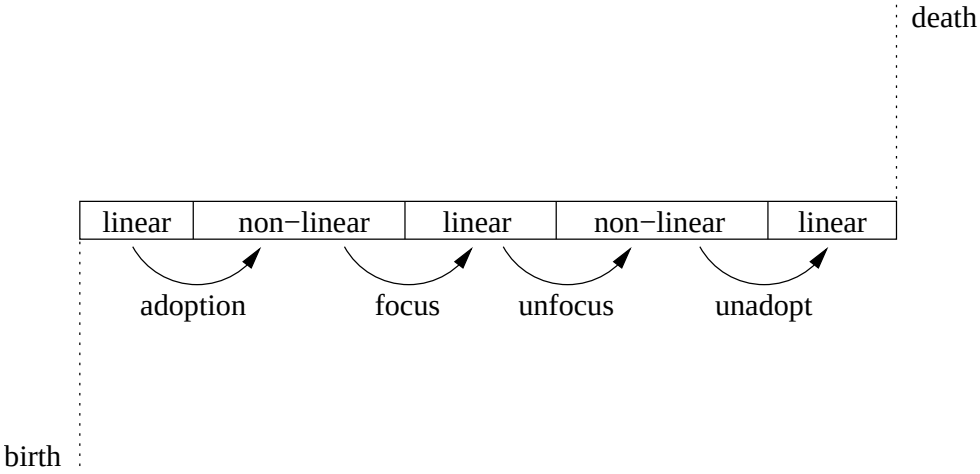


Figure 4: Using adoption and focus to switch types

As we can see one uses adoption and focus for the transition from linear to non-linear and vice versa. Vault implements tracked types and implicitly keys that protect linear types and guards for non-linear types. If a key `k` is accessible then all normal operations are allowed on the linear type. Similarly, if we hold all the guards for the non-linear type then all normal operations are allowed. The operations to convert from one type to the other are called adoption and focus.

Since all objects are created as a linear type and since it is impractical to program without aliasing Vault uses adoption as a way to obtain aliases.

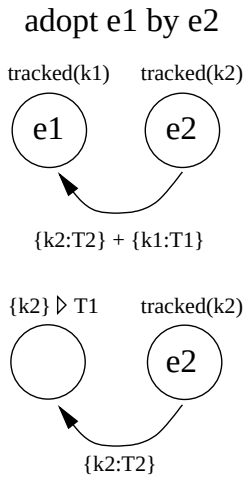


Figure 5: An example of adoption

Figure 5 describes the process of an adoption: `adopt e1 by e2` which takes an adoptee, say `o1` which is the result of `e1` and an adopter, say `o2` which is the result `e2` both of linear type and consumes the linear reference to `o1` by creating an internal reference from `o2` to `o1`. An adoptee has thus exactly one single

adopter and the result of the adoption expression is a reference to a non-linear type to the adoptee. This adoptee's nonlinear type is then its previous linear type with only one top-level type constructor changed from linear to nonlinear. Access to `o1` is disallowed through the internal reference of the adopter and the non-linear reference to the adoptee is valid for the entire lifetime of the adopter. Then, any linear components of the object `o1` can not be directly accessed through the non-linear reference since that would lead to a shared access violation to objects of linear type. However, the linear components of `o1` may be accessed in the scope of a focus operation. When the adopter is disposed, all non-linear references to the adopted object become inaccessible. Adoptions stack so one can adopt several objects through multiple adoption expressions.

Focus provides a way to temporarily view an object of non-linear type as a linear type. The turning point is that any type invariant of a non-linear object can be violated as long as no alias for the object can witness the violation.

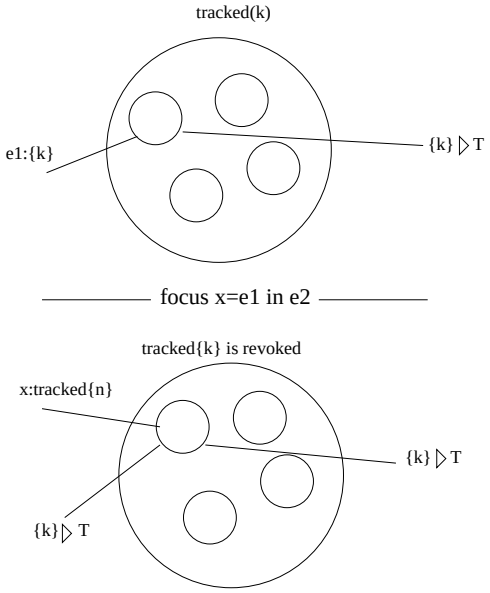


Figure 6: An example of focus

The focus construct `x=focus e1 in e2` requires `e1` to evaluate to an object of a guarded type. This object is called the focus object. The first thing to do is to bind `x` to the focused object and give `x` a tracked type `tracked{n}` for a newly created key. Now, because `x`'s tracked type, the expression `e2` can change the type associated with the key `n` and can thus access and replace linear components. Second, in order to ensure that no aliases can witness these changes, within the context of `e2`, the original guarding key is removed. In doing so one guarantees that no aliases of the focused object are accessible during `e2`. The focus is ended by revoking the temporary key `n`. The following is a code example of how this would be done:

```
void resize({D} cell c)
```

```
{
   focus x=c in
      free x.data;
   x.data = new array;
}
```

3.3 Hoare and Separation Logic

3.3.1 Hoare Logic

Hoare logic is a formal system invented by C. A. R Hoare and offers a way to reason about computer programs and program flow at a mathematical level. Hoare does this by introducing the "Hoare triple" which, in its simplest form is described by a pre-condition, a command and a postcondition. This concept can also be observed in token automation systems, and relies on the fact that any action or command has an initial state and an end state. If the command does not terminate, there is no end state (which appears later on as we shall see when we analyze goto labels). Thus we can say that the simplest Hoare triple written in the form:

$$\{P\}\,\mathtt{C}\,\{Q\}$$

is a formal way of saying: "whenever `P` holds in the state prior to the execution of `C` , then `Q` will hold afterwards or `C` doesn't terminate".

- Empty statement

$$\overline{\{P\}\,\mathtt{skip}\,\{P\}}$$

The empty statement can be seen, from the perspective of functionality, as a `NOP` -similar operator. It basically states that nothing is executed and the precondition becomes directly the postcondition of the next statement without any change.

- Assignment axiom

$$\overline{\{P[x/E]\}\mathtt{x} := \mathtt{E}\{P\}}$$

Basically this is saying that if `P[x/E]`, meaning the expression in which all occurrences of the non-bound variable `x` have been replaced by the expression `E` , held before the assignment, then it will also hold for the postcondition.

- Conditional rule

$$\frac{\{P \wedge Q\}\mathtt{C}_1\{R\}, \{\neg P \wedge Q\}\mathtt{C}_2\{R\}}{\{Q\}\mathtt{if}\,P\,\mathtt{then}\,\mathtt{C}_1\,\mathtt{else}\,\mathtt{C}_2\,\mathtt{fi}\{R\}}$$

The conditional rule allows us to model typical `if` statements found in all programming languages. Note that the final state, represented by `R` is identical after the conditional as well as the original state `Q` being the same before the conditional.

- While rule

$$\frac{\{P \wedge Q\}\mathtt{C}\{R\}}{\{P\}\mathtt{while}\, Q\, \mathtt{do}\, \mathtt{C}\, \mathtt{done}\{\neg Q \wedge R\}}$$

The while rule lets us model loops. One important feature is that at the end of a loop, the invariant, here illustrated by `R` will remain unmodified whereas the loop condition `Q` will be false. Judgment on these two is a means to show that the loop is not endless which is a very frequent programming error.

- Rule of Composition

$$\frac{\{P\}\mathtt{C}_1\{Q\}, \{Q\}\mathtt{C}_2\{R\}}{\{P\}\mathtt{C}_1\mathtt{C}_2\{R\}}$$

This rule allows us to entail several commands in a sequence, the starting state being `P` , the "intermediary" state being `Q` and the final state being `R` . This rule is important since it shows us that it is applicable to state transitions commonly found in programming languages.

- Consequence rule

$$\frac{P' \rightarrow P, \{P\}\mathtt{C}\{Q\}, Q \rightarrow Q'}{\{P'\}\,\mathtt{C}\,\{Q'\}}$$

3.3.2 Separation Logic

Separation logic[36] is a substructural logic and an extension of Hoare logic attributed to John C. Reynolds describing a way of reasoning about programs. Particularly, separation logic addresses several problems where other logics become hard to understand. Similar to the mix of intuitionistic and linear logic concepts, separation logic tries to solve certain issues pertinent to context splitting. Thus, separation logic facilitates reasoning about programs that manipulate data structures[32], ownership transfers and modular reasoning between concurrent modules and brings the concept of local reasoning.

Most important programs make serious use of the heap (apache, tcp/ip, etc..) yet heap verification, as we have seen, is particularly hard. With separation logic though, we can address the problem by splitting the heap into several parts and reasoning locally on these parts. Using classical logic, separation logic adds a few other constructs a few of which we mention:

- `emp` meaning the heap or the heaplet is empty
- `x` $\mapsto$ `y` meaning that the heap or heaplet has exactly one cell `x`, holding `y`
- `P` $*$ `Q` the separating conjunction, meaning that the heaplet can be diveded so `A` is true for one part of the heap and `B` for the other.
- `P-*Q` the separation implication which means that if we extend the current heap `P` we will have a heap that satisfies the assertion `Q`

To illustrate the key difference between the separating conjunction and the AND operator we use two pointers:

$$x \mapsto 3, y * y \mapsto 3, x$$

which asserts that `x` points to an adjacent pair of cells containing `3` and separately a pointer `y` that points to an adjacent pair of cells containing `3` and `x`. For clarity we may represent this as:

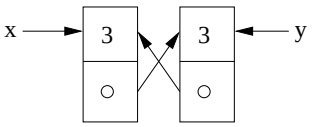


Figure 7: x points to two cells, the last cell pointing to y and y points to two cells, the last cell pointing to x

On the contrary, if we would have used the AND operator instead of the separating conjunction, as in:

$$x \mapsto 3, y \wedge y \mapsto 3, x$$

then both `x` and `y` would reside in the same heap and could only happen if `x` and `y` would hold the same values.

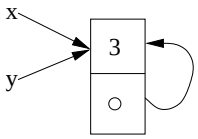


Figure 8: x and y are in the same heap and can only happen if x and y are the same

Sooner or later we need to use the frame rule that allows us to expand our assertions `P` and `Q` to include statements that describe parts of the heap not affected by a command as we shall see later on in the concurrency[6] example. The frame rule is simply:

$$\frac{\{P\}\mathtt{C}\{Q\}}{\{P * R\}\mathtt{C}\{Q * R\}}$$

where there are no free variables in `R` modified by the command `C`.

Separation logic also introduces two new constructs, `new` and `dispose` which lets us reason about creation and disposal of objects. These two constructs are essential when reasoning about pointers and memory.

The precondition for `new` is `emp` since before allocation the heap is empty. After the `new` command, the newly created pointer exists and points to a block of memory in the postcondtion:

$$\{emp\}\mathtt{new(k)}\{k \mapsto -\}$$

Similarly, the `dispose` command takes a pointer and frees the block so that we obtain `emp` in the postcondtion:

$$\{k \mapsto -\}\mathtt{dispose(k)}\{emp\}$$

Separation logic has a wide ranged application in lists and has been used on many occasions to reason about them. A typical application could be reasoning about device drivers[9, 2] since these make terse use of lists. However these examples use separation logic for code safety and consistency rather security.

3.3.3 Vault and Cyclone

Both dialects introduce the concept of regions where the heap may be seen as an entity composed of several disjunct parts. A region is simply a named subset of the heap. We can use the concept of splitting up the heap into several dijunct sub-heaps to illustrate the implementation of regions.

In Vault objects are allocated individually in a region and deallocated as a whole. An interface, as previously discussed implements regions in Vault and is given by the following code:

```
interface REGION {
type region;
tracked region create();
R:byte[] alloc(tracked(R) region reg, int size)[R];
void delete(tracked(R), region reg)[-R];
}
```

Since code safety is Vault's primary goal, a region is tracked so one would spot the accidental misuse of the region. Consider the following example where a region containing a string is disposed before the actual assignment to the string:

```
tracked(R) region reg = Region.create();
R:String st = new(reg) String "Hello World";
if(st.equals("Hello World")
{
   Region.delete(reg);
   st = "Bye World!";
}
```

Such a construct will yield an error since the region has been previously deleted and Vault report the misuse of memory.

One could represent a region in Vault using the following illustration in Figure 9:

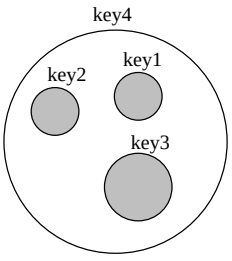


Figure 9: A region is "tracked" by a key and contains objects, themselves "tracked" by keys.

showing that a region can contain multiple sub-regions each tracked by individual keys which could be expressed in separation logic as: `key1` : `A` $*$ `key2` : `B` $*$ `key3` : `C`

given that we name the subregions `A`, `B` and `C` respectively. Each region is tracked by a certain key and when a region is deallocated its key is discarded.

Cyclone employs a similar strategy, by doing region analysis. For example, consider the following code in C:

```
char *itoa(int i)
{
   char buf[10];
   sprintf(buf, "%d", i);
   return buf;
}
```

This function returns an object that is allocated on the stack of the function `itoa` which is not available after the function returns. Compilers such as gcc will warn about the code yet the following will compile without any warning:

```
char *itoa(int i)
{
   char buf[10], *p;
   sprintf(buf, "%d", i);
   p = buf;
   return p;
}
```

Cyclone does regional analysis of each segment of code in order to prevent dangling pointers like the one returned from the latter version of our function `itoa`. All local variables in a scope are considered to be part of the same region which is separated from the heap or any other local regions. Compiling the latter version in Cyclone would make it report an error once it observes that `p` is a pointer into the local stack.

4 Using Logics to Enforce code safety

4.1 A Buffer Overflow Example

Suppose we model the allocation of an array similar to the one shown previously in the example above. Reasoning about the pointer `s` we can say that before allocation, the heap is empty and after the allocation, the pointer would point to the memory block. It is irrelevant of what type the pointer is, or what exactly it is pointing to thus we can write the simple array allocation rule:

$$\{emp\}\,\mathtt{x} = \mathtt{new(k)}\left\{\overset{k-1}{\underset{i=0}{*}} x + i \mapsto -\right\} \text{ where } \overset{k-1}{\underset{i=0}{*}} \mathtt{x_i} = \mathtt{x}_0 * \mathtt{x}_1 * ... * \mathtt{x_k}$$

The precondition suggests that the heap is empty before the allocation, and the postcondition shows the allocated block of memory after the allocation as a conjunction of separate cells. Given this allocation, whenever a specific cell is being accessed, we can check whether that cell has been previously allocated just by judging on the indexes similar to common array manipulation.

Suppose we want to access the cell index `j` and consider `j` an arbitrary address value. If the address value is within the memory block in which our pointer `x` points, then, it can be considered a valid call. If it is not within bounds then the call would be invalid and would allow us to overwrite memory which wasn't specifically preallocated. Since we're reasoning on indexes, a simple if-then-else rule could do the bounds checking. Given the if axiom in Hoare logic we can write the following:

$$\frac{\left\{ \overset{k}{\underset{i=0}{*}} \mathtt{a{+}i} \mapsto - \wedge \mathtt{j} \leqslant \mathtt{k} \right\} [\mathtt{a{+}j}] := \mathtt{m} \left\{ \overset{k}{\underset{i=0}{*}} \mathtt{a{+}i} \mapsto - \right\}, \left\{ \overset{k}{\underset{i=0}{*}} \mathtt{a{+}i} \mapsto - \wedge \mathtt{j} > \mathtt{k} \right\} \mathtt{skip} \left\{ \overset{k}{\underset{i=0}{*}} \mathtt{a{+}i} \mapsto - \right\}}{\left\{ \overset{k}{\underset{i=0}{*}} \mathtt{a{+}i} \mapsto - \right\} \mathtt{if\ j} \leqslant \mathtt{k\ then\ [a{+}j]{:=}m\ else\ skip\ fi} \left\{ \overset{k}{\underset{i=0}{*}} \mathtt{a{+}i} \mapsto - \right\}}$$

Which basically means we're inferring that if the new requested address is within bounds, then accept the call and set the contents of that address to a desired value `m`. If it is not in the address pool, do nothing and refuse to set the new address.

4.2 An Integer Overflow Example

One other type of overflow that can be observed in attacks, is the integer overflow which occurs when an arithmetic operation attempts to create a numeric value that is larger than what can be stored in the given storage type. This usually triggers some relatively "undefined" behavior. Some processors will simply saturate the value returning the maximum possible result of the storage type while others will wrap storing the least significant bits of the result. In C for example, signed integer overflow would cause "undefined behavior" while unsigned integer overflow would reduce the number modulo of two wrapping around on overflow.

Consider the following code which illustrates a classic integer overflow scenario:

```
int a, b, result;
result = a + b;
```

this will yield "undefined" behavior if the integers a and b contain the maximum values that integers could store. The integer "result" will not be able to store the sum of a and b since it can itself hold only the maximum integer value.

There are several security implications based on the choice of behavior. Suppose that in some situations a program would test if a variable contains a positive value. in case this value has a signed integer type, an overflow would cause the value to wrap and become negative thus changing the test condition and overriding the program's logic. The same would happen to negative values wrapped to positive values. Combining what we have learned before on buffer overflows, we can for example presume that a buffer is allocated using a memory allocation function of a size depending on an integer operation. In case this operation is manipulated and made to wrap around, it could lead to allocate a buffer of arbitrary size and leading further to a buffer overflow attack.

In order to circumvent such attacks, a simple method can be devised to check whether a value can be stored in a specific type. We define, for purposes of clarity, the types `int8_t`, `int16_t`, `int32_t` and so on in which the numbers represent the number of bits that can be represented. In this case, we can transform the previous piece of code in something more precise:

```
int8_t a, b, result;
int16_t temp;

temp = (int16_t) a+b;

if(temp<2^7-1 && temp > -2^7) {
   result = (int8_t) temp;
}
else {
   skip;
}
```

By upcasting the result of the sum to a larger storage type, the arithmetic can be performed. Further to that, in order to keep the code consistent, we check if this temporary result is between the lower and upper bounds of the smaller storage type. If that is the case, then the result can be stored in the smaller type and the arithmetic is not vulnerable to an integer overflow. Knowing that, we can model our logic to express the check. We suppose that type `int8_t` is a subtype of `int16_t` which in turn is a subtype of `int32_t` and so on. Then we can express the former program using Hoare logic:

$\{\exists a, b, result \in int8\_t \wedge \exists temp \in int16\_t \wedge int8\_t \subset int16\_t\}$

we then upcast the addition:

$\texttt{temp} = (\texttt{int16\_t})\texttt{a} + \texttt{b};$

and following the IF axiom for Hoare logic:

$\{temp < max(int8\_t) \wedge temp > min(int8\_t) \wedge temp = a + b\}$
$\texttt{result} = (\texttt{int8\_t})\texttt{temp};$
$\{result < max(int8\_t) \wedge result > min(int8\_t)\}$
$\{\neg(temp < max(int8\_t) \wedge temp > min(int8\_t)) \wedge temp = a + b\}$
`skip`
$\{result < max(int8\_t) \wedge result > min(int8\_t)\}$

thus inferring that:

$\{temp = a + b\}\texttt{if}\ temp < max(int8\_t) \wedge temp > min(int8\_t)\ \texttt{then}\ result = (int8\_t)temp;\ \texttt{else}\ error\ \texttt{fi}\ \{result < max(int8\_t) \wedge result > min(int8\_t)\}$

and thus letting the assignment to proceed in case there wouldn't be any overflow or fail in case the sum would exceed the maximum storage of variable `result`. Note that this implementation, like our previous example concerning buffer overflows, uses a `skip` command denying to perform the overflow but continuing the execution flow. This is optional. It may throw an error and try to fail gracefully but it is up to the implementation to decide what action

should be taken in case the process is a critical application. We prefer to detect the error and leave it up to the user to implement some sort of handling as this could go many ways.

4.3 A Concurrency Example

One other type of attack brought to our attention by Zalewski in his underground paper [43] on signal handlers relies on resource sharing and synchronization between threads[4]. It is clear that in low level languages resource sharing between threads has been an issue since there is no practical definition of what can be accessed at a certain time by two or more concurrent processes. It is also clear that the conflict arises when two distinct processes or threads, to be more precise, can share resources but shouldn't access a resource at the same time. This would lead to a collision specially when the shared resource is modified in some way. Suppose you have a file writing operation going on in one process and another file writing operation going on in another process. If these two processes acquire a lock on a resource at the same time, what would be written to the file? A good answer is that its contents can not be determined precisely. We wouldn't know how to distinguish between the write of the first process and the second write of the second process. For this, certain methods have been introduced to assure some safety, or at least some logic, in such issues. These issues make up a broad class of problems called concurrency[6] which, in our specific case, vaguely fall into the category of race conditions primarily due to the fact that a certain logic race can be observed between the first and the second process trying to access the same resource; in the worst case, at the same time. This leads to the necessity to define two different concepts in low level programming languages such as C:

- Re-entrance safety
- Thread safety

The first one, re-entrance safety tries to make repeated calls to a function safe. Suppose a pointer is freed once in one pass of the function. If the function is called again, the same pointer would be freed leading to a crash. This leads to a number of conditions which make a function re-entrant safe:

- The function must not hold any static non-constant data.
- The function must not return the address to a static non-constant data.
- It must work only on the data provided by the calling function (the function arguments).
- It must not rely on locks to singleton resources.
- It must not call non-reentrant functions.

If a function satisfies this property then it can be considered a safe re-entrant function. If we take the example of IO functions, we can observe that these rely heavily on shared singleton resources, for example storage disks. This definition narrows down a whole set of functions to a bare sum of very few functions which

may be considered re-entrant safe. Thus, it would be convenient to find a way to modify a non-reentrant function so that it can be considered re-entrant safe. For example, we could rule out those functions which return pointers to data defined statically within the function body and refuse to compile it at compiler level. Consider a simple example:

```
char *global_pointer;
...
void f() {
...
    free(global_pointer);
...
}

int main(void) {
...
    global_pointer = (char *) calloc(10, sizeof(char));
...
    f();
    f();
...
}
```

The main function would do two subsequent calls to the `f` function thus resulting in a double free corruption. The `f` function operates on the global pointer thus making it non-reentrant safe. In this simple case, we can find a way around it by simply defining a global variable and using Hoare logic. Consider the following modification to the code:

```
char *global_pointer;
int flag=0;
...
void f() {
...
    if(flag==0) {
        free(global_pointer);
        flag++;
    }
...
}

int main(void) {
...
    global_pointer = (char *) calloc(10, sizeof(char));
...
    f();
    f();
...
}
```

This modification would make our code re-entrant safe since the flag would be incremented after the freeing of the global pointer and a subsequent call to the function would fail the test. But suppose that we are running on a multi-core machine and that the f function is actually the initializer function for two distinct threads. More precisely, the unpublished paper on signal handlers shows us a small variation on the example code we gave above:

```
char *global_pointer;
int flag=0;
...
void f() {
...
    if(flag==0) {
        free(global_pointer);
        flag++;
    }
...
}

int main(void) {
...
    global_pointer = (char *) calloc(10, sizeof(char));
...
    signal(SIGHUP, f);
    signal(SIGTERM, f);
...
}
```

By attaching a signal handler to two different signals we can obtain some sort of concurrency, in that the function f may be called at the same time by two different threads under a multi-core environment. Looking at the code, we concluded that the vulnerability was partially due to bad programming because the same signal handler function was bound to two different signals which would make the program call the signal handler in two very different circumstances. The easy remedy for this would have been to handle each signal handler function to different signals and make the re-entrant safe in order to avoid potential pitfalls such as a double free. However, this solution would require the programmer to change the code and that may not be a feasible solution specially when dealing with large code in which case it would be unlikely and relatively hard to determine when such an event occurs and under what circumstances. Consider the illustration in Figure 10 of what would actually happen.

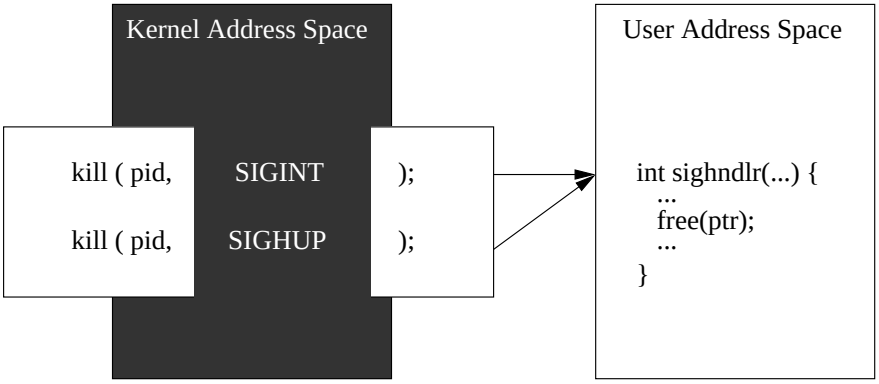


Figure 10: Delivering signals and causing a double free memory corruption.

In that case, flag may hold the value `0` across several threads at the same time resulting in a double (or multiple) free no better than without the flag variable modification. This brings us closer to concurrency and threads. A piece of code can be considered thread-safe if it functions correctly during simultaneous execution by multiple threads. Thus, the latest variation on our example can be considered re-entrant safe but not thread safe since the latter would lead to a double free again. In order to make functions thread safe the two most common solutions are to introduce two different programming mechanisms:

- Semaphores
- Mutexes

both of which are guaranteed to be atomic. Semaphores compare slightly to the variable `flag` we have been using and they are meant to be arrays of flags which the programmer may use in order to make sure that a resource or a block of code is executed only by one thread at a time. Similarly, mutexes provide mechanisms for effectively locking down variables so they may only be accessed by one thread at a time.

Using flags and mutexes we can ensure both the reentry safety and the thread safety of our example code:

```
char *global_pointer;
int flag=0;
...
void f() {
...
    if(flag==0) {
        pthread_mutex_lock(global_pointer);
        if(flag == 0) {
           free(global_pointer);
           flag++;
        }
        pthread_mutex_unlock(global_pointer);
```

```
    }
...
}

int main(void) {
...
    global_pointer = (char *) calloc(10, sizeof(char));
...
    signal(SIGHUP, f);
    signal(SIGTERM, f);
...
}
```

We have to note that the mutex is unlocked only after the flag has been set. If this would not be the case, access to the flag could have been shared at the same time by two different threads. Also the flag is checked again once the lock is acquired in order to avoid freeing the same pointer again by the next signal once it has acquired the lock.

Following the article [12] we can model mutexes in pre- and post conditions. The notations is not standard but the article describes the mutex lock as follows: A lock takes a location `x` whose `lk` field is zero and replaces it with `TID` (the tread identifier) thus acquiring the lock specific to the current thread. This is later released by the unlock function which sets `TID` to `0` thus allowing any other thread to acquire the lock.

We can thus model our thread and re-entry safe method using Hoare and separation logic in the following manner:

$\{x \mapsto lk = 0 * flag = 0 * ptr \mapsto -\}$
`if(flag == 0) {`
$\{x \mapsto lk = 0 * flag = 0 * ptr \mapsto -\}$
`pthread_mutex_lock(ptr);`
$\{x \mapsto lk = TID * flag = 0 * ptr \mapsto -\}$
`if(flag == 0) {`
$\{x \mapsto lk = TID * flag = 0 * ptr \mapsto -\}$
`dispose(ptr);`
$\{x \mapsto lk = TID * flag = 0 * emp\}$
`flag:=flag`$+1$;
$\{x \mapsto lk = TID * flag = 1 * emp\}$
`}`
`pthread_mutex_unlock(ptr);`
$\{x \mapsto lk = 0 * flag = 1 * emp\}$
`}`
$\{x \mapsto lk = 0 * flag = 1 * emp\}$
$\{x \mapsto lk = 0 * flag = 1\}$

$\{x \mapsto lk = 0 * flag = 0 * ptr \mapsto -\}$
`if(flag == 0) {`
$\{x \mapsto lk = 0 * flag = 0 * ptr \mapsto -\}$
`pthread_mutex_lock(ptr);`
...
...
...
...
...
...
...
...
...
$\{x \mapsto lk = TID * flag = 1 * emp\}$
`if(flag == 0) {`
`}`
$\{x \mapsto lk = TID * flag = 1 * emp\}$
`pthread_mutex_unlock(ptr);`
$\{x \mapsto lk = 0 * flag = 1 * emp\}$
`}`
$\{x \mapsto lk = 0 * flag = 1 * emp\}$
$\{x \mapsto lk = 0 * flag = 1\}$

and by specifying the pre and post conditions of every operation. In the example above it is important to state that `phtread_mutex_lock` also performs the `P/V` operations required for threadding which is included in the POSIX thread library[31]. That is, `pthread_mutex_lock`, beside being guaranteed to be atomic (ie: even in our worst case scenario only one thread will be able to acquire the lock) it also blocks operations until the lock is freed. In other words the `pthread_mutex_lock` operation will wait for the lock to be freed before entering execution of the successive commands. This behavior can be changed by altering flags (as per the manual page of the POSIX threadding library[8]) however the default operation is to block the thread until the lock is released. Another thing to watch out for would be a possible deadlock in case the first thread does not release the lock, however, given the logic we propose above, in all circumstances and final postcondition, the lock is released allowing the second thread to acquire the lock in turn.

The main drawback of this method is that a global flag must be used or, in C terminology a static variable which will pertain its value across several function calls. Since we can use the power of static variables in C, it would seem feasible enough to implement the proposed solution by introducing a static variable within the function body which will keep its value until the program terminates. This would eliminate the need to introduce a global variable throughout the whole program and may reduce memory consumption which is not trivial if we think about how Cyclone and Vault comparatively double the needed of resources of a ported program.

Other alternatives would have been to set the pointer to `NULL` and test for `NULL` in the precondition. This would be inadvisable since setting the pointer

to NULL in the function would not necessarily set the pointer to NULL globally. To illustrate this, we use a simple example:

```
void bar(char *ptr)
{
   ptr = NULL;
   if(ptr == NULL)
      printf("Yes, ptr is locally: %x in bar() but ", ptr)
}

int main(void)
{
   char *ptr;
   ptr = (char *) calloc(5, sizeof(char));
   ptr = "Horia";
   bar(ptr);

   if(ptr != NULL)
      printf("string \"%s\" is at: %x\n", ptr, ptr);

   return 0;
}
```

Thus a sample run of the snippet would result in:

```
Yes, ptr is locally: 0 in bar() but string "Horia" is at: 8048506
```

So unless ptr is a pointer declared in a global scope, the function will just set its local value to NULL not producing the expected results. Because of this, it is unfeasible to just set the pointer to NULL and we need a global flag as a record of the deallocation of ptr and that's also why one would prefer to opt for the logic variant using mutexes and flags described above. Of course, this may be implemented as a semaphore yet the overhead is too much just to make the function reentrant safe. It is however plausible to create a semaphore to store an array of flags for a whole program doing deallocations making it easier to track memory at any point in the program. Also, using a global variable would allow us to reason about the pointer later on when the NULL variant will not. We have thus seen that by using a global flag and a mutex lock we can guarantee the safety of signal handler problem.

4.4 Extending Mutex Locks

System call wrappers provide means by which the kernel security model can be extended and allows certain software to intercept system calls and alter their behavior. This is common, for example, in various anti-virus software packages which intercept the file writing operation in order to check data before it is commited to the store. Other uses include auditing or locking down services in a tinder-box by validating the effects on the entire system before certain operations are executed.

Furthermore, for added security, system call wrappers implement a pre- and postcondition mechanism to make sure that certain requirements are validated on entry and exit. One such implementation is the Sysjail wrapper, intended to protect processes with superuser privileges and places itself between the kernel and the processes by validating or rewriting system call arguments, if necessary, to keep the process confined. A network precondition of the Sysjail wrapper is that it will only accept the binding address to be `INADDR_ANY` or the local address of the tinder-box. However, this assumes that the precondition check and the `bind()` system call are atomic which is far from being true. It actually opens up an opportunity for a timed-attack between the validation state and the actual binding to the address.

Operating system kernels are highly concurrent and make use of shared memory in order to allow communication between the various threads and to allow processes to communicate between userland and kernel space. A system call wrapper is one such example which is located between the userland and the kernel space like reference monitor presumably adding a new level of security to the whole system. This opens up a whole set of problems which can lead to race conditions, deadlocks, data corruption and makes logical reasoning about paralelism quite challenging.

It is thus possible[40], under a concurrency context, that system call interposition may lead to contrary effects and even prove to be a security hole rather than provide means to enforce it in the first place. The cited paper shows that attacks on the system wrapper itself give way to privilege escalation, possibilities of overriding auditing processes and down right to practical exploiting techniques by using a classical TOCTOU (Time of check to time of use) attack which is possible due to the fact that certain operations are non-atomic and thus, given a certain time-frame, the wrapper may allow the orignal data to be overwirtten by a concurrent process. Two other techniques, which we don't mention, are described as derviates of the former in which the attacker forges audits and overrides system call arguments before the kernel accesses them.

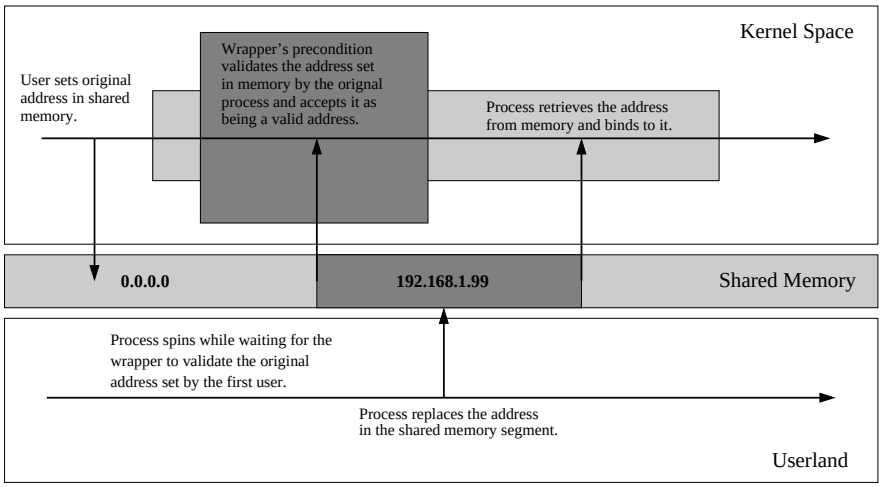


Figure 1: A classical TOCTTOU attack in which the attacker replaces a

validated address with a new one after the system wrapper's precondition has already validated the orinigal address.

One attack, depicted in Figure 1, relies on the fact that the validation process and the actual binding to the address are not atomic processes and can't thus guarantee that no data corruption has occurred inbetween. In the case presented above, the second process has replaced the address in the shared memory segment after validation and before the actual `bind()` system call. Watson's paper carefully describes and implements the correct spinning technique which the second process must undergo before it can actually override the shared memory segment and replace the validated address with an arbitrary address.

The obvious question that comes to mind in such an example (and also aplicable to all other examples in Watson's paper) is why doesn't the user lock down the address in the shared memory? One reason, as stipulated by Watson is that locking down the address one might violate the essence of concurrency and may give rise to a number of problems since there might be other concurrent processes which might rely on the address in the shared memory segment in order to serve various purposes. However, and as the scope of this document implies, it is unclear why any other concurrent process should be able to override that address. Since the original `INADDR_ANY` address was specified, it is unlikely that the user changed his mind between the calls or that he would want any process to replace it with something else. Nevertheless, it is feasible that a process, for example's sake, an auditing process, may want to read that address in order to execute and there is no reason why the memory segment should be locked down for reading.

We propose an extension to the `mutex_lock()` implementation which would allow us to set fractional permissions, in this case, read and write permissions, on a locked resource in order to provide transparent reads of the address in the shared memory segement and to assure that the address can't be overwritten even though nor the wrapper or the `bind()` system call are atomic operations.

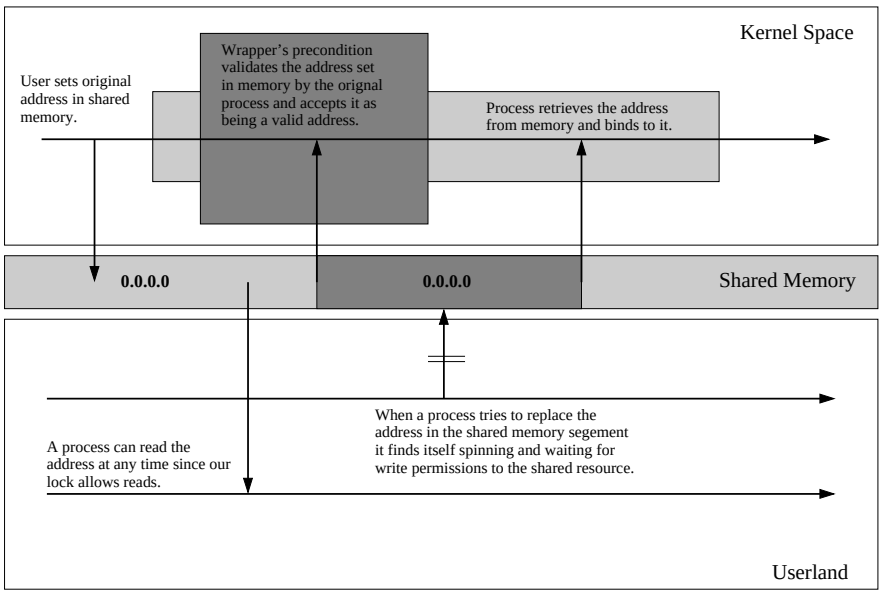


Figure 2: Writes to the shared resource are disallowed and the attacking process finds itself spinning and trying to acquire a write lock which won't be released until the `bind()` *system call copies the address from shared memory. Reads are however allowed by our lock and any process may access the shared resource with a read request.*

We suggest a modification to the actual locking mechanism since it will only require minor modifications and, in our opinion, would work transparently without needing to rewrite whole parts of the kernel itself. By splitting a lock and allowing fractional permissions we also extend the semantics of locking down shared resources. Original functionality will also be preserved and restored by simply setting, and respectively unsetting the fractional permissions on the shared resource thus emulating the original mutex lock mechanism. It may also allow better resource management given the fact that at any time a locked down resource may have different properties assigned.

We can sketch a rough diagram of our proposed model with the code segements relevant to our modifications.

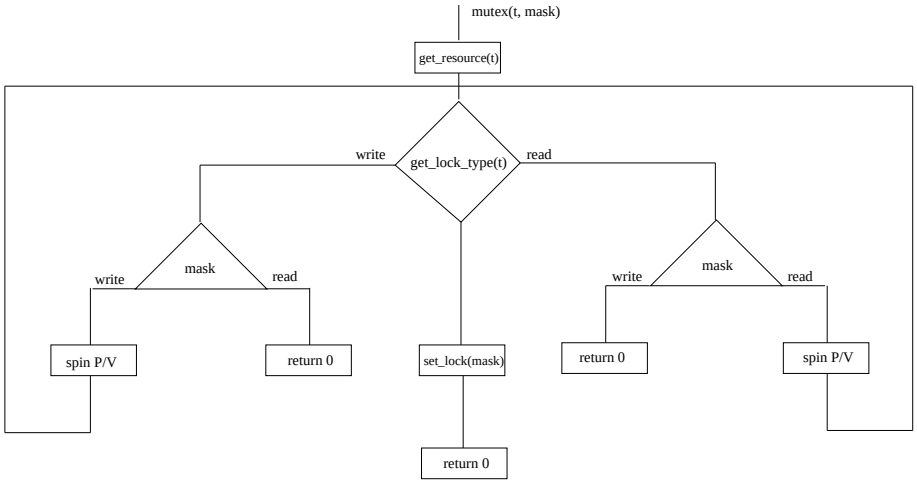


Figure 3: Trying to acquire a write lock on a resource which has been already locked to writing will spin until the lock has been released. The same applies when trying to acquire a read lock on a resource that has been locked down for read.

5 Proposed Work

As we have seen, we can employ different logics to describe various events that govern a programming language or a program itself. Not only can we describe events but we are also able to enforce behavior through static checks or prevent unwanted situations through run-time tests. The ongoing research, employing these logics has been mostly geared towards code safety rather than security as we have seen in the case of separation logic being used for bug detection in device drivers. A minor part, almost nonexistent refers to the area of security such as the case of Cyclone and Vault. However, that part has, out of several reasons, become either outdated or abandoned by the developers. Parts of it though and concepts which we can draw from past work can be employed in future work, such as our work, in which we seek to not only find a better way in representing the events governing a programming language or a program but also in its practical implementation as a whole modular construct. It is our conclusion that Cyclone and Vault both employ a strategy of avoiding security pitfalls through limitation and restriction rather than, what we think would be best, transparent logical reasoning. Most of Vault's safety implementation can be loosely regarded as access control structures whereas Cyclone enforces the use of the dialect by restricting access to normal operations (such as pointer arithmetic). We find that it is imperative that a language or dialect should not restrict the use of typical constructs or types and force the programmer to reason differently about the code since by doing so it limits the capabilities of the language as a whole. Unlikely, yet possibly, a programmer might want or need to run code which is not entirely safe for a particular reason and given the former mentioned tools that is not possible (we could give as an example kernel programming where sometimes it is desirable to drop down to inline assembler in order to program certain parts). A different approach, as we beg to differ,

would be to (a) understand what the code is meant to do, (b) implement checks for that purpose and disallow misuse and lastly (c) transform code (similar to what CRED does) rather than imposing new types and new constructs upon the user. The following Figure 11 vaguely depicts the system we're proposing:

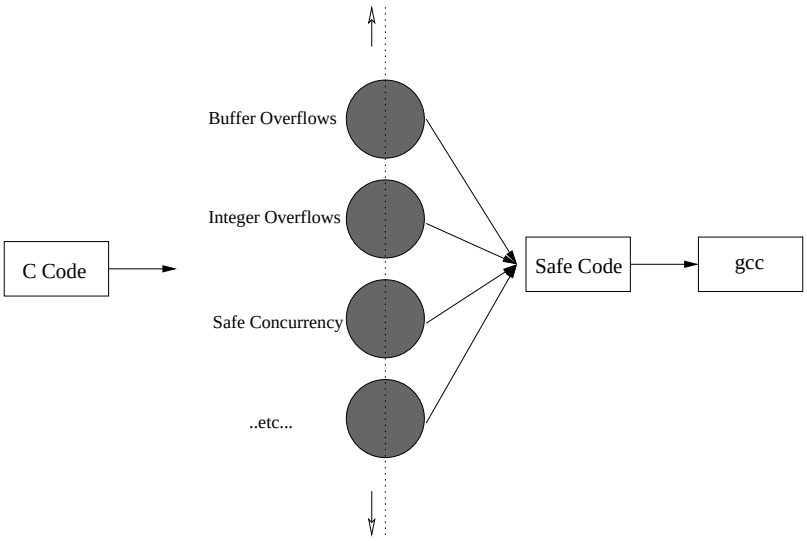


Figure 11: C code is the input to our wrapper which applies corrections and fixes to the code transforming it into "safe code" which, in turn, can be compiled by the standard gcc compiler.

The first thing we can observe is that similar to Cyclone's system, we won't be building a whole compiler from scratch since that would greatly exceed the proposed time frame. Also, it is undesirable to rewrite what has and is constantly written by a large number of people (in our example, the gcc compiler). However, we propose a modular system in which a certain number of independent sub-modules can target a certain problem which we can approach either by doing active research on current attacks or by deriving them from common problems we have mentioned. These modules will be completely separate and will not require a specific order of implementation so that the resulting work will be fully functional even if it lacks a sub-module.

The final product will be a wrapper that takes standard C code and transforms it by adding static or run time checks based on the logic we're studying and transforms it into what we call "safe code" which can then in turn be used as input for the gcc compiler. Our choice of compiler is due to the fact that gcc is Open Source and we can thus trace errors down quickly and even though choosing Vault as the final compiler would in some ways reduce the workload we still prefer to use gcc. This will also imply that the generated safe code will be compatible with standard C (pertaining to ISO C may be a choice) so that it will in the end compile. We consider this to be an important part in our work since we want to use logic reasoning in our work rather than access control concepts such as we have observed in Vault or forcing new types like we observed in Cyclone. It is desirable for our wrapper to attempt and warn effectively about unsafe constructs (like CCured allows) rather than throwing

an error and refusing to compile the unsafe code. This will also be important for what we have previously mentioned since it is sometimes desirable to write unsafe code.

Additionally, we desire to focus on the security aspect of the language rather than its code safety and this can be achieved by actively researching common attacks and the particular vulnerabilities attackers exploit in code. One good example, followed by a counter-example is the signal handler attack where the code can be considered safe and will compile flawlessly on Cylone yet the logical reasoning shows us that there might be a problem with double free-ing the pointer in a concurrency context. The counter-example is, of course, Vault which implements a very strict and rigorous access control system without showing much interest to security but rather code safety. As such, the individual modules should focus, in our view, on security even if it implies digging deep into various key areas of research, say concurrency for example, rather than implementing a syntactic construct of some sort. The latter is, in our view, more related to code safety rather than security yet overlaps between the two concepts will be allowed as long as the primary goal of the module is to approach a security vulnerability with logical reasoning.

Given our modular approach, our wrapper will be Open Source and will benefit from the advantages of a collaborative project. A decent goal for this wrapper would be the ability to use our wrapper and prevent a certain vulnerability within a selected piece of code. Analysis or verification though will just be a part of the module as we want to take it one step further and actively "fix" the relevant part of code or construct. Warnings will also be allowed and implemented similar to what we have seen in Splint. In this context, the goal is to have something functional which can, and most importantly, will be used rather than an obscure language construct or formal debates about the vulnerability itself. Normally, if the logic reasoning behind our "fix" is sound, then the code should follow and turn it into reality. Another advantage is that by making our wrapper modular it will allow it to be cumulative and even if part of the work is done, it will still be functional in the end and wouldn't require or rely on some bits which weren't previously realized. Of course, some work may and will be unfinished yet it is our consideration that partial security is better than no security at all.

During development it is probable that our system will use different types of logic systems. For example, we can use separation logic to reason about memory consistency or use linear types to prevent multiple accesses to the same resource. It is clear that we must combine several concepts to circumvent errors in programs as it is feasible to say that we can not rely on one single type of logic to build our entire system based on it. This is obvious since even separation logic uses Hoare logic constructs like pre- and postconditions but also introduces context splitting through the separating conjunction $*$. Similarly, it is not excluded that we may need to employ linear types to reason about resources in the fashion Vault does region handling. The important thing to mention is that it won't be one solid block implementing a new type of logic or a language in itself yet it would be more of an environment similar to Splint which would not only detect problems but also fix them automatically before the resulting "safe code" can be used as input to the standard gcc compiler.

Additionally, we propose a level-based system to our wrapper which can be loosely seen as the optimization flags which gcc uses. To explain this, we refer to Figure 12 which shows the internal breakdown of the levels and what these represent in our system:

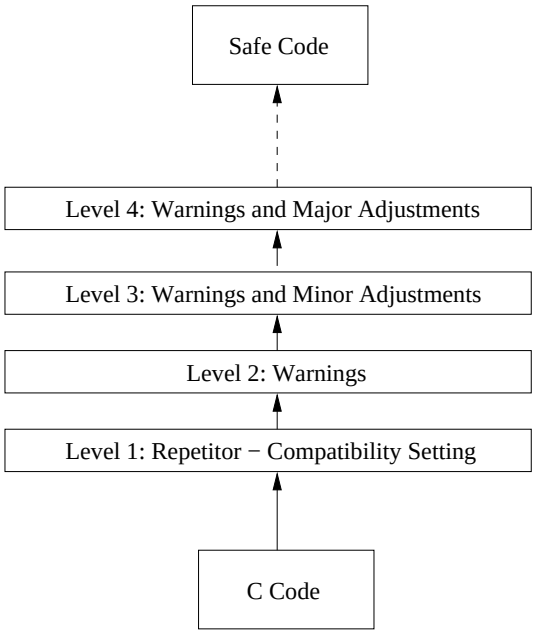


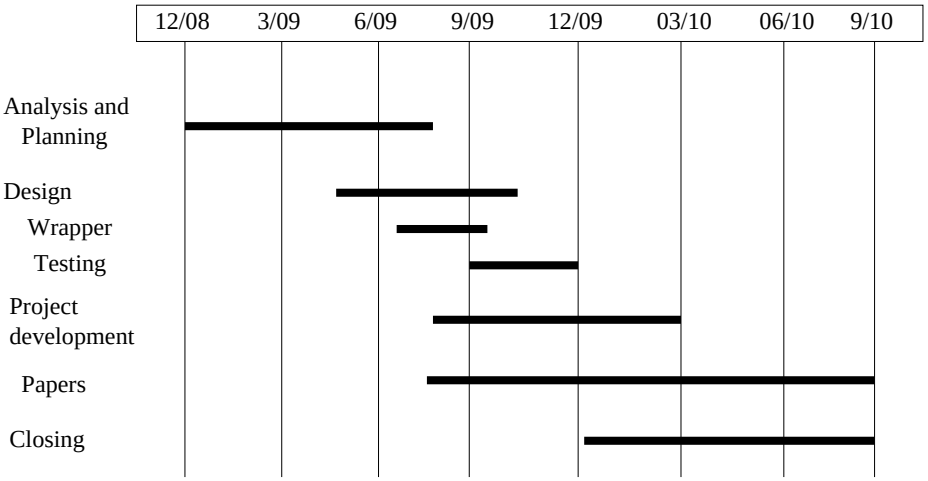
Figure 12: Level-breakdown of the proposed wrapper spanning multiple levels which affect the code in various ways.

C code will get parsed by our wrapper and then, using the different modules it will attempt to fix the issues which it finds. However, we find it necessary to break these fixes into what we denote as being "levels". The very first level, and probably the most important one in terms of compatibility is the pass through repetitor level in which code passes unaltered through our wrapper and is taken by gcc to be as it is. One the second level, our wrapper should warn about mistakes it finds in the code it took as input. This is just the verification layer we've been talking about previously where the code is analyzed but not altered in any way in which the wrapper functions similarly to Splint. The third level would for example, fix very obvious problems just minorly altering the output and the last one would apply all the knowledge of our modules altering all the code if necessary. We find this proposed functionality (loosely similar to Cyclone pass through declarations) to be necessary in order to deal with compatibility issues as well as allowing our wrapper to be more versatile in terms of functionality (ie: it can be run just as a code verifier without altering code, partially altering code and applying all fixes and modifications in order to remedy the code). In terms of compatibility it is clear that the first entry level would be the most compatible, guaranteed not to break anything and the last level the least compatible.

6 Strategy

The main purpose of our tool will be to verify code or modify code to fix certain security flaws. If our tool indicates that a part of code is safe then it will be logically provable that that certain part of code is safe. By using Hoare logic we will be able to specify, through annotations, the pre- and postconditions of certain blocks of code. If our tool will accept them then it will mean that the tripple is derivable in Hoare logic hence satisfying the soundness of the tool. We might have to employ linear logic when we reason about keys or effects specific to regions or fractional permissions in concurrency contexts. Separation logic is essential to this research since it provides the possibility to split the heap in disjoint parts and reason about memory individually allowing us to model what has already been done previously in Cyclone or Vault.

We conclude with an estimated time table to illustrate the different steps we will take while working as well as the interaction between the several stages of our project.



- **Analysis and planning:** Taken as a whole, this is the initial stage where we plan what the project will do exactly (ie: will it be more than a parser? Will it be a stand-alone tool or will we prefer to release patches to the C compiler itself? The preferred method is, of course, a self standing wrapper since it will be easily maintained but it is not excluded that we might submit a patch upstream to gcc dev incorporating our check(s)). The various sub-modules illustrate what this chapter will include:
  - **Requirements** consist of what we will need in the development of the project and there might be a few since if this is to be a collaborative project we will need CVS/SVN access and probably a testbed machine we can run tests on.
  - **Planning** comes in with the general outline of the project (ie: how the different levels will be implemented in the wrapper). Also, this part will include formalizing concepts, for example concurrency in

Hoare/Separation logic so we can have a precise definition of every concept before we go ahead and implement the modules.

- **Design:** This stage is important since we need to determine the overall structure of the project and the underlying modules and how these will interact in the final product.
  - **Wrapper:** In this stage we will write the general mainframe of the wrapper maybe at level 1 as previously seen in Figure 12. At this stage we might decide whether we will split it into patches and a self-standing tool.
  - **Testing:** This stage is essential and will require creating a small prototype to test the interaction between the inputs and outputs of the wrapper (ie: how it interacts with gcc). It will also provide the basic framework and allow us to build our modules on top of it.
- **Project development:** This will be the core of the project and will imply writing all modules and additions to the wrapper in the previously created framework. It is closely related to the papers since we'll be conducting ongoing research and studying attacks and how to prevent them while incorporating new modules for our wrapper. Upstream patches to gcc development aren't excluded at this stage.
- **Papers:** This time is allotted to writing papers and we can expect quite a few since we are actively researching security vulnerabilities and adding modules at the same time to our wrapper. Certainly one foreseeable paper will be on the internals of our system, how it behaves and a comparison with other alternative projects which attempt the same thing. Also we will mention how it differs from other projects (say Splint for example) which focus on code safety rather than security.
- **Closing:** Writing the thesis. At this point the project is stable enough to start writing the thesis in parallel with other activities. Total success of the project would mean that we would have implemented all the modules and we have a working wrapper. Total failure would mean that we didn't manage to implement any module maybe not even the wrapper itself. Due tot the modular design of the project, we can have partial success in which a number of modules have been implemented and we have solved some issues. Note that the project can be ended at any time since it is not a requirement that a specific module, or component, of the project must be implemented. It might be that a certain module attracts more attention than others and if our research indicates that it is worth investigating further since it represents an interesting class of problems (ie: concurrency) the project may grant focus to that specific module.

References

[1] A. Barber and G. Plotkin. Dual intuitionistic linear logic. LFCS Report Series - Laboratory for Foundations of Computer Science, 1998.

[2] J. Berdine, C. Calcagno, B. Cook, D. Distefano, O'Hearn, P.W. Wies, and T. Yang. Shape analysis for composite data structures. *Lecture Notes in Computer Science*, 4590:178–192, 2007.

[3] J. Berdine, C. Calcagno, and P. W. OHearn. Smallfoot: Modular automatic assertion checking with separation lgic. *FMCO*, 2006, 2006.

[4] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *32nd POPL*, pages 59–70, 2005.

[5] A. Bratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. *Proceedings of 2000 USENIX Annual Technical Conference*, pages 18–23, 2000.

[6] S. Brookes. A semantics for concurrent separation logic. *Lecture Notes in Computer Science*, 3170:16–34.

[7] BUlba and Kil3r. Bypassing stackguard and stackshield. *Phrack*, 0xa, 2000.

[8] David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[9] C. Calgano, D. Distefano, O'Hearn, and P.W. Yang. Footprint analysis: A shape analysis that discovers preconditions. *Lecture Notes in Computer Science*, 4634:402–418, 2007.

[10] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wangle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks, 1998.

[11] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. pages 227–237, 2003.

[12] Matthew J. Parkinson Cristiano Calcagno and Viktor Vafeiadis. Modular safety checking for fine-grained concurrency. *SAS*, 2007:123–134, 2007.

[13] Trevor Jim Dan Grossman, Michael Hicks and Greg Morrisett. Cyclone: a type-safe dialect of c. *C/C++ Users Journal*, 23(1), 2005.

[14] Robert DeLine and Manuel Fahndrich. Enforcing high-level protocols in low-level software. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 56–69, 2001.

[15] Fahndrich and M. DeLine. Adoption and focus: Practical linear types for imperative programming. ACM Sigplan Notices:13–24, 1999.

[16] M. Fahndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in singularity os., 2006.

[17] M. Fluet, G. Morrisett, and A. Ahmed. Linear regions are all you need. *Proceedings of the 1st ACM SIGOPS EuroSys Conference*, pages 7–21, 2006.

[18] Jeffrey Scott Foster. *Type Qualifiers: Lightweight Specifications to Improve Software Quality*. PhD thesis, University of California, Berkeley, 2002.

[19] John V. Guttag and James J. Horning an Jeannette M. Wing. The larch family of specification languages. *IEEE Software*, pages 24–36, 1985.

[20] C. A. R. Hoare. Am axiomatic basis for computer programming. *Communications of the ACM*, 12(576-585), October 1969.

[21] C. A. R. Hoare and J.C. Shepherdson. *Mathematical Logic and Programming Languages*. Prentice-Hall International Series in Computer Science, 1985.

[22] G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An overview of the singularity project. *Microsoft Research: Techinal Report*, 135, 2005.

[23] G. Hunt and R. Larus. Singularity design motivation. *Microsoft Research: Technical Report*, 105, 2004.

[24] S. C. Johnson. Lint, a c program checker. *Murray Hill, N.J. : Bell Telephone Laboratories*, 1977.

[25] Y. Lafont. The linear abstract machine. *Theoretical Computer Science*, 59(1-2):157–180, 1988.

[26] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. *Proceedings of the 2003 Network and Distributed System Security*, pages 123–130, 2003.

[27] Patric Lincoln. Linear logic. *ACM SIGACT Notices*, 23:29–37, 1992.

[28] P.D. Magnus. *forallX*. University at Albany, State University of New York.

[29] George C. Necula, Scott McPeak, and Wes Weimer. Taming c pointers. urlwww.cs.berkeley.edu/ necula, 2004.

[30] George C. Necula, Scott Mcpeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy code. *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 37:128–139, 2002.

[31] Bradford Nichols, Dick Buttlar, and Jacqueline P. Farrell. *Pthreads Programming*. O'Reilly, 101 Morris Street, Sebastopol, CA 95472, 1998.

[32] O'Hearn, P. Reynolds, and J. Yang. Local reasoning about programs that alter data structures. *Lecture Notes in Computer Science*, CSL:1–19, 2001.

[33] Aleph One. Smashing the stack for fun and profit. *Phrack*, 49, 2001.

[34] Hilarie Orman. The morris worm: A fifteen-year perspective. IEEE Security and Privacy:35–43, 2003.

[35] David J. Pym. On bunched predicate logic. Logic in Computer Science, 14th Symposium, 1999.

[36] J. Reynolds. Separation logic: a logic for shared mutable data structures. *LICS*, '02(29), 2002.

[37] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector. *Proceedings of the Network and Distributed System Security*, pages 159–169, 2004.

[38] J. Seward and N. Nethercote. Valgrind: A framework for heavyweight dynamic binary instrumentation, 2007.

[39] P. Wangle, Perry, C. Cowan, and Crispin. Stackguard: Simple stack smash protection for gcc. *Proceedings of the GCC Developers Summit*, pages 243–256, 2003.

[40] Robert N. M. Watson. Exploiting concurrency vulnerabilities in system call wrappers. *Proceedings of the first USENIX workshop on Offensive Technologies*, 2007.

[41] Westley Weimer. *The CCured type system and type inference.* Berkely: Computer Science Division, University of California, 2003.

[42] Chris Wright, Crispin Cowan, and James Morris. Abstract linux security modules: General security support for the linux kernel, 2002.

[43] M. Zalewski. Delivering signals for fun and profit. 2001.

[44] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using cqual for static analysis of authorization hook placement. *Proceedings of the 11th USENIX Security Symposium*, pages 33–48, 2002.