Imperial College London

MEng Individual Project

Department of Computing

Imperial College of Science, Technology and Medicine

# A C Memory Model for use with Concolic Testing

*Author:*
Charles Loveman

*Supervisor:*
Daniel Schemmel

June 15, 2024

## Abstract

Concolic Testing is a form of automated program testing that combines concrete testing with symbolic execution to perform a guided search over the possible inputs to a program. For automated testing to detect a bug it needs an error oracle, which is a program that can detect if a bug has occurred during an execution.
We implement an error oracle for violations of a memory model into a prototype Concolic Testing tool called CCF. This enables the detection of some memory bugs that occur in C programs. We design a memory model to classify what behaviours are considered memory bugs, drawing from similar memory models implemented for other Symbolic Testing tools like KLEE.

# Contents

# Chapter 1

# Introduction

The annual cost incurred by software vulnerabilities increases year on year[22]. There is a wide range of possible vulnerabilities, but one of the most commonly exploited classes of vulnerabilities are memory bugs. These bugs occur when the program violates the rules that govern how a program may interact with memory, called the memory model of the programming language, and these bugs can be very difficult to find.

C and C++ have been used to implement many high performance systems. In order to achieve the best performance, the language standards for C and C++ state that the result of violating many memory rules is undefined behaviour. This makes it difficult to determine if the memory model has been violated during execution as the program could have show any behaviour after the rules are broken. As a result, the class of memory models that describes C and C++ is particularly vulnerable.

Due to the inherently dynamic nature of memory operations in this memory model, we cannot verify statically if a program violates the memory model. Instead, we need to implement a dynamic system to check if the model has been violated at runtime. This is similar to the behaviour of memory sanitizers like ASAN [24], which instrument the target program so that when it is run the memory errors are reported.

Despite the existence of tools like ASAN which can detect memory errors, manual and randomised testing campaigns can still miss bugs in programs. ASAN only reports an error for a single run of the program. This means that without a method to determine the best inputs to try, the number of bugs found will be limited. Concolic execution tackles this problem by recording the constraints that had to be met on the input to the program to take a particular path through the program and uses an SMT solver to find new inputs which take different paths through the program. This gives us an automated method to increase test coverage and generate new inputs which could detect bugs.

For concolic execution to construct these constraints, we need to store metadata regarding the operations that have been performed on the values of the program during execution. As we already store metadata about program values, this can be extended to add support for a memory model checker which will validate each memory access the target program performs. We design and implement a memory model checker into a prototype concolic execution framework called CCF.

# Chapter 2

# Ethical Issues

As a computing project, this project utilises computers which can have a negative impact on the environment. However, we expect that the benefit gained by detecting program errors will outweigh the environmental impact of running the software. The software produced will make use of open source software, which will not infringe on their copyright.

As concolic execution software can find bugs in programs, a malicious attacker who obtained access to a victims source code could use the software to find vulnerabilities in the victim's software which would allow them to cause harm. However, as the program is publicly available to the victim also, they should be able to find and fix this vulnerability with the help of the program and it is worth noting that the software will not create any new vulnerabilities, only expose ones that already exist.

# Chapter 3

# Background

Every programming languages has a specification defining what behaviour is and isn't valid. Despite this, some bugs can exist in code that conforms with the language specification. To determine if an error has occurred during execution, we need an error oracle. For automated software testing we need an automated error oracle, or we won't be able to determine what is a bug. As a result, many bugs cannot be found with automated testing.

This definition of a set of rules regarding memory interactions for a programming language is called a memory model and a violation of these rules is a memory bug. We can construct an automated error oracle based on the set of rules to determine if a memory bug has occurred. This makes memory bugs a suitable target for automated testing.

We can define the class of memory models used by many programming languages including C and C++ as follows.

$$AddressSpace ::= \mathcal{N} \rightarrow Byte \tag{3.1}$$

We define the address space of a program to be a function from a positive integer to a byte (3.1).

$$MemoryObject \subseteq \mathcal{N} \tag{3.2}$$

Although these languages may have many different types of objects, we define the representation of the object in memory as a memory object (3.2). A memory object is a set of memory addresses and each memory object must be a subset of the domain of the address space.

$$LiveObjects \in \mathcal{P}(MemoryObject) \tag{3.3}$$

$LiveObjects$ is the set of memory objects allocated at a point in execution (3.3). $LiveObjects$ is a disjoint set, as memory objects cannot overlap. We can fully define the state of memory at a program point with the pair $(AddressSpace, LiveObjects)$.

It is important to note that memory objects have no internal structure and cannot be defined recursively. On most architectures, the set of memory addresses allocated to a memory object will be contiguous, although we do not require this for the memory model.

$$AccessRange \subseteq \mathcal{N} \tag{3.4}$$

$$Alloc ::= (AddressSpace, LiveObjects, AccessRange) \rightarrow LiveObjects \tag{3.5}$$

$Alloc$ adds a new memory object to the set of live objects. The new memory object will equal the address range provided, so we require that $LiveObjects \cup AccessRange$ is disjoint.

$$Dealloc ::= (AddressSpace, LiveObjects, AccessRange) \rightarrow LiveObjects \tag{3.6}$$

$Dealloc$ removes a memory object from the set of live objects. The access range specifies the object to be removed, so we require that $AccessRange \in LiveObjects$.

$$Access ::= (AddressSpace, LiveObjects, AccessRange) \rightarrow \mathcal{B} \tag{3.7}$$

Every memory access must occur inside of a live memory object, so

$$\exists x \in LiveObjects.\, AccessRange \subseteq x$$

Bytes in memory can either be allocated, which means they are an element of exactly one live memory object, or they are unallocated, which means they are not contained in any memory object.

## 3.1 The C Memory Model

A program can only have a bug if we define a set of rules that specify what behaviour is legal and what is not. A memory model is a formal description of the memory store and operations over it [19]. As the C standard does not provide an executable memory model checker, there are a variety of interpretations of the C memory model which have been proposed.

In C, an object is an instantiation of a type or an allocation returned from a memory allocator. To avoid confusion we will call this a C-object. C-objects can be defined recursively, allowing a program to create a C-object that contains other C-objects. This is used in arrays, structs, and unions to organise data. This is different to memory objects which only have a location and a set of bytes; memory objects do not have internal structure or types.

In C, the period in which an C-object is guaranteed to be stored and its value is valid to access is called its lifetime. The lifetime of an object depends on the C-object's storage duration. The C standard defines four storage durations: static, thread, automatic, and allocated [14]. Static C-objects are created before the program starts and their lifetime continues until the program terminates. Thread C-objects are created when a thread is created and will exist until the thread terminates. A C-object with automatic storage duration will exist until the block in which it is created ends.

Finally, if a C-object has allocated storage duration it will exist until the memory region containing the object is explicitly freed. We understand that the memory object associated with the C-object must be resident in memory while the C-object is alive.

For an implementation to be considered a C memory model it must therefore ensure that any access to an object occurs within that object's storage duration. This suggests an obvious implementation: whenever an object is allocated store some additional metadata regarding the objects storage duration and whenever the object is accessed ensure it is within the stored duration. We can model this behaviour using the general memory model by representing the start of the storage duration as an allocation and the end of the storage duration as a deallocation. Therefore, we can focus on C without losing the generality of the general memory model.

What don't we need? pointer arithmetic, provenance, etc

## 3.2 Checking a program for violations of a Memory Model

The AddressSanitizer (ASAN) [24] is a dynamic memory model checker, which means that memory bugs can only be reported when the program is executed. As ASAN needs to modify the internal behaviour of the system under test (SUT), it instruments the SUT at compile time to insert behaviour whenever the SUT accesses memory. This allows it to monitor all memory interactions of the program and check if the memory model is violated.

The memory model used by ASAN is similar to the one described above. This model does not store memory object metadata inside of the memory objects themselves. In fact, given only a set of bytes there is no way to determine if the bytes are allocated or unallocated, or what memory object they are allocated to. As a result, ASAN stores additional metadata about every allocated byte so that it can verify if the memory model has been violated.

To check a memory model we need to monitor the memory allocations, deallocations, and memory accesses. When memory is allocated we need verify that all the bytes to be allocated are currently unallocated. As the memory model does not allow the user to request the addresses at which bytes are allocated this issue is generally avoided. On an allocation, ASAN will update its metadata to mark the bytes as allocated.

When memory is deallocated we need to check that all the bytes belong to the same object. ASAN accomplishes this by checking its metadata to see if all the bytes are allocated and ensuring that the memory objects are spaced apart with redzones to check if the deallocation spans between memory objects.

When memory is accessed we need to check that all the bytes belong to the same object. ASAN accomplishes this in the same way as with deallocations. All we need to check the memory model is to store metadata for each object and then check any memory accesses against the stored metadata.

## 3.3 Finding More Inputs

Although ASAN is a powerful tool for reporting memory model violations, it only works if the program input provided triggers a memory bug to occur. The number of possible inputs grows exponentially with the size of the input data, so the difficulty of finding inputs that trigger bugs also grows exponentially.

A single program execution can be described by which direction is taken at each branch encountered. This is called a path in the program. When testing we would like to cover all paths in the program, however, the number of paths grows exponentially with the number of branches. This means covering every path is often infeasible.

The majority of program testing is done using manually written tests [12]. We can write test programs which execute part or all of the SUT with set input parameters, to examine the behaviour of the program and ensure that the execution performs as expected. Testing has many advantages, it is easy to set up and all modern programming languages will have extensive support for writing test cases. Furthermore, tests can be written as the program grows and develops to create a suite of tests that describes the intended behaviour of the program. However, testing has a key problem, where as the program grows the number of possible execution paths through program can grow exponentially. As a single test only follows one path through the program, the programmer would need to manually write an exponential number of test cases to cover the full behaviour of the program. As a result, test suites for real software will not cover every possible path, potentially allowing bugs to remain in the program.

Randomised testing was developed to build upon manual testing by automatically generating test cases. Also called Fuzzing [21], testing programs with random inputs has been able to find many bugs in long running software projects like GCC [30, 32]. The effectiveness of random testing comes from the ability to search a greater number of paths compared to manual testing and a random test can avoid the biases that the test author may hold and search the whole input space fairly. However, in a large program some paths will only be reached from a small section of the input space and randomised testing finds paths in proportion to how many inputs take the path. This means that random testing is unlikely to find bugs on low probability paths.

An alternative to random testing is concolic execution, which uses a guided search to find new inputs. Concolic execution has a higher overhead compared to random testing so the proportion of the input space searched will be lower but this trade off is made in favour of finding inputs that should take new program paths. This means that concolic execution should explore a new path with every input tried, although implementation details means this may not always occur in practice.

## 3.4 Concolic Execution

Suppose that we are interested in testing to see if the program in Figure 3.1 can crash and through random testing we have tried the input $arr = (-5)$, $i = 0$ and

```
int clamp(int x) {
    if (x < 0) {
        return 0;
    }
    if (x > 5) {
        abort();
    }
    if (x > 10) {
        return 10;
    }
    return x;
}
```

**Figure 3.1:** This program accesses the variable $x$ and clamps the value between 0 and 10. However, if the value is greater than 5 then the program will crash.

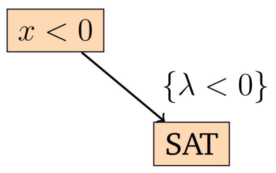
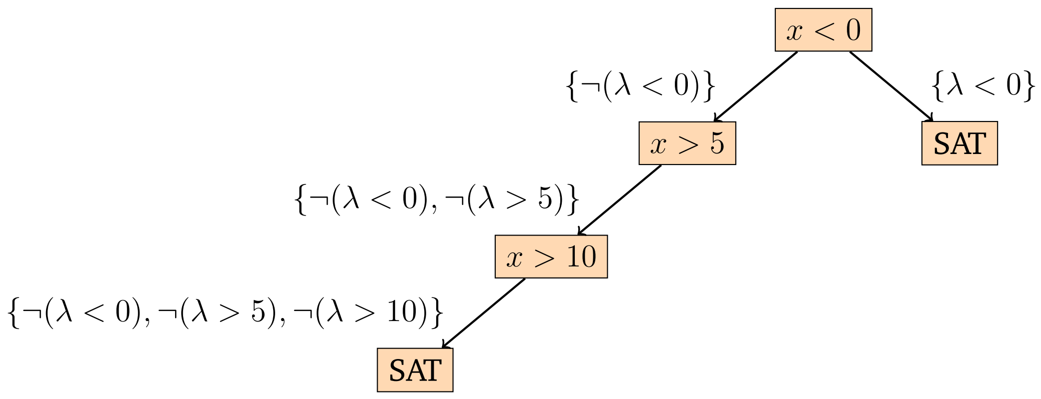


**Figure 3.2:** Execution paths in clamp function after 1 execution, with $x = \lambda$.



$x < 0$
$\{\neg(\lambda < 0)\}$
$\{\lambda < 0\}$
$x > 5$
SAT
$\{\neg(\lambda < 0), \neg(\lambda > 5)\}$
$x > 10$
$\{\neg(\lambda < 0), \neg(\lambda > 5), \neg(\lambda > 10)\}$
SAT

**Figure 3.3:** Execution paths in clamp function after 2 executions, with $x = \lambda$.

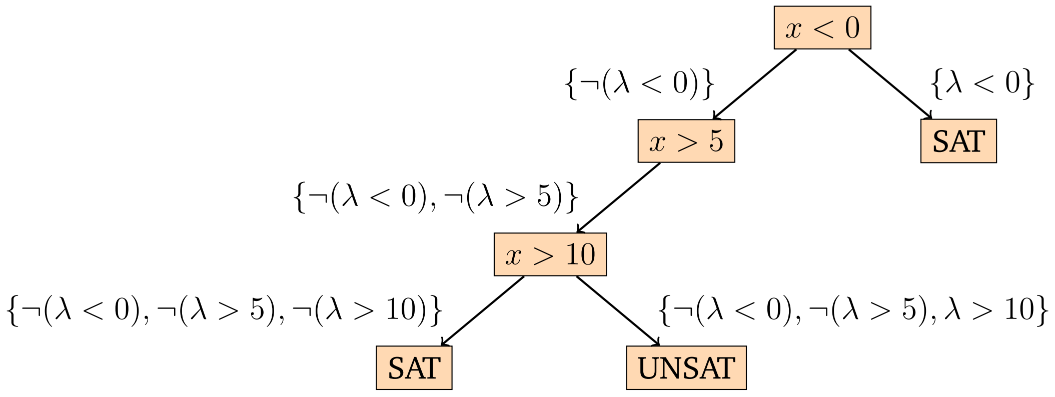

$x < 0$
$\{\neg(\lambda < 0)\}$
$\{\lambda < 0\}$
$x > 5$
SAT
$\{\neg(\lambda < 0), \neg(\lambda > 5)\}$
$x > 10$
$\{\neg(\lambda < 0), \neg(\lambda > 5), \neg(\lambda > 10)\}$
$\{\neg(\lambda < 0), \neg(\lambda > 5), \lambda > 10\}$
SAT
UNSAT

**Figure 3.4:** Execution paths in clamp function after 2 executions and 3 SMT queries. The third query returned UNSAT.

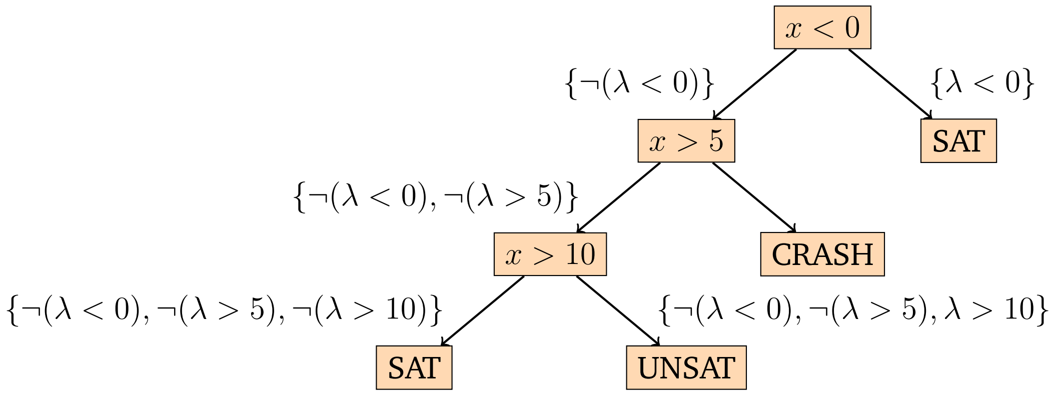

$x < 0$
$\{\neg(\lambda < 0)\}$
$\{\lambda < 0\}$
$x > 5$
SAT
$\{\neg(\lambda < 0), \neg(\lambda > 5)\}$
$x > 10$
CRASH
$\{\neg(\lambda < 0), \neg(\lambda > 5), \neg(\lambda > 10)\}$
$\{\neg(\lambda < 0), \neg(\lambda > 5), \lambda > 10\}$
SAT
UNSAT

**Figure 3.5:** Execution paths in clamp function after 3 executions and 4 SMT queries. The final execution ends in a crash, so we have found a bug.

found that the program did not crash and returned 0. If we were only doing random testing then the next step would be to pick completely random values of $x$ and hope that we could get a different result. Concolic execution gives us a method to take the results of previous executions to select more interesting inputs to try.

We assign a symbolic expression to represent the value of the inputs of the program. For example, let $x = \lambda$, where $\lambda$ could take any integer value. Then we run the program with a concrete input, such as $x = -5$. Whenever a variable is assigned we additionally compute a symbolic expression to represent its new value and when the execution branches we record the conditions required to take the branch as a constraint on the symbolic values.

When we reach the branch $x < 0$, we can evaluate the condition concretely to determine if the branch will be taken. As $-5 < 0$ is true, the branch will be taken. Additionally, we record the constraint $\lambda < 0$. This means that in order to take this path through the program, the input must satisfy the expression $\lambda < 0$. We call the sequence of constraints recorded for a single path the path constraint.

We can generate an input which takes a new path by taking the path constraint and negating the final constraint. Here we only have one constraint so the new path constraint is $\{\neg(\lambda < 0)\}$. We can use a Satisfiability Modulo Theories (SMT) solver to find an input satisfying the constraint. It could return any number greater than or equal to 0, lets take 3 for this example.

After executing the program with the input $x = 3$, we get the path constraint $\{\neg(\lambda < 0), \neg(\lambda > 5), \neg(\lambda > 10)\}$. Negating the final constraint gives us $\{\neg(\lambda < 0), \neg(\lambda > 5), \lambda > 10\}$, but when we query the SMT solver it will give us the result UNSAT. This means there is no input which takes that path through the program. Figure 3.3 shows the paths explored in an execution tree. To find the next input we query the SMT solver for any non-empty prefix and negate the final constraint of that, such as $\{\neg(\lambda < 0), \lambda > 5\}$. Executing the new input should find the crash in the program.

This process is called concolic execution and is a type of symbolic execution [31]. Concolic is a portmanteau of Concrete and Symbolic, referring to how in concolic execution variables have concrete values and symbolic expressions. We have seen how concolic execution can be used to efficiently construct and explore the execution tree of a program.

Symbolic execution was developed in the 1970s [15, 16, 10, 2] as a way to test programs. 'Symbolic' means that the inputs given to the SUT do not have a single concrete value but instead each input can take a set of possible values. For example, let $x = \lambda$, where $\lambda$ is a symbolic value with possible values $1, 2, 3$. Then after executing $x = x + 1$, $x$ will equal the symbolic expression $\lambda + 1$ with the possible values $2, 3, 4$. In this way all the operations of a programming language can be extended to act on symbolic values.

### 3.4.1 Creating Constraints

To find new inputs we need to generate the path constraint for the path taken by the program. This is done incrementally, creating a constraint at each branch on symbolic data. Assuming there is no non-determinism, branches on purely concrete data will always take the same branch outcome for a single path in the program. As a result, we don't need to create constraints for branches on concrete data.

We associate a symbolic value with each input to the program. If we let $x \leftarrow \lambda$ and then execute $y = x + 1$, we need to store $x \leftarrow \lambda, y \leftarrow \lambda + 1$ to describe the symbolic state. We can then construct the constraint using the condition expression of the branch and the symbolic data. For example, if we branch on $if(y > 2)$, we will create the constraint $\lambda + 1 > 2$.

As variables can be associated with symbolic values, the address of a memory access may also be associated with a symbolic value. We call the symbolic value associated with an address a symbolic address. We want to find symbolic inputs that cause a memory access to a symbolic address to access every memory object that could be addressed by the symbolic address. Symbolic execution accomplishes this by forking over all possible objects. Similarly, in concolic execution we need to create a constraint on the symbolic address.

### 3.4.2 Solving Constraints to Find New Inputs

Once we have the path constraint, we can negate the final constraint on the path and query a Satisfiability Modulo Theories (SMT) solver to find an input that satisfies the new path constraint, if such an input exists [1].

SMT solvers extend Satisfiability (SAT) solvers to allow the use of background theories for greater expressibility. A SAT solver [8] takes a logical formula and determines if there is an assignment to the boolean variables in the formula that will result in the formula being true. As SAT solving is NP-complete [7], solving SAT and SMT queries is inherently hard. As a result, for real programs we cannot guarantee that the solver will return a result in a reasonable amount of time. This means some paths of the program may end up unexplored.

SAT solvers only work with boolean variables, which means we would need to represent numbers with one variable per bit and manually implement logic for arithmetic and logical operators on these numbers. Using an SMT solver allows us to work with numbers much more easily, as SMT solvers provide the theory of Quantifier-Free Bitvectors which allows us to describe program behaviour using operations on bitvectors. We have to use the Quantifier-Free theory as introducing quantifiers makes the theory undecidable.

An important theory for modelling the use of memory is the array theory [3] which can be used to model changes to an array of values. This is used by symbolic execution tools like KLEE [5] to create constraints on memory accesses.

### 3.4.3 CCF

CCF is a new concolic executor that aims to improve on existing concolic executors by efficiently executing C programs concolically and providing support for external function calls. This should allow for a greater variety of C programs to be executed which could promote the use of concolic execution on real programs. As CCF is a

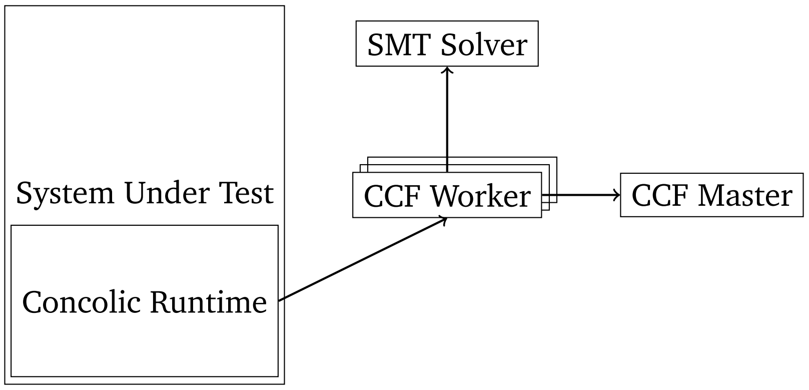


**Figure 3.6:** The processes CCF uses for concolic execution. The master orchestrates multiple workers. Each worker performs concolic execution by executing instances of the system under test. The concolic runtime is embedded in the system under test and tracks the required metadata.

prototype it has some features that are yet to be implemented. One example of this is a memory model, so for this project we will implement a memory model.

CCF is a complex project composed of many sub-systems. Figure 3.6 shows the components used for concolic execution.

To track the metadata required for both concolic execution and memory model checking, CCF first instruments the target program to add function calls into the Concolic Runtime. The Concolic Runtime maintains the metadata for a single program execution and is updated by the instrumented functions in the target program. When a branch occurs during execution, the instrumented program calls a function in the runtime to report the branch. From this the runtime can compute the constraint for the branch based on the branch condition and the symbolic data stored and can send the constraint to the CCF Worker.

The CCF Worker performs concolic execution of the system under test. To do this it performs the full concolic execution loop, of running the target program, collecting constraints from the runtime, queries the SMT solver with the negated path constraint, and then repeating with the new input found by the SMT solver. The worker builds the execution tree from the path constraints collected and uses this to explore the program.

The CCF Master orchestrates the execution of multiple workers. This provides a single interface to the user while allowing parallel concolic execution.

To implement the instrumentation of the target program, CCF uses the LLVM infrastructure. LLVM is also used by other symbolic execution tools like KLEE [5] and allows the definition of custom compiler passes. This makes it possible to implement program instrumentation while taking advantage of existing compiler implementations like Clang. This greatly reduces the work needed to create a working C compiler.

As CCF allows the SUT to call uninstrumented external functions, the behaviour of the program under test may be non-deterministic. This is a problem for concolic execution, as it means that an input that satisfies the path constraint for a particular program point when executed may take a different execution path. We have no way to guarantee that we have explored all program paths after non-determinism has occurred and there is also no guarantee that if CCF finishes execution the program is bug-free. Non-determinism is detected when the path constraint from a new input does not match the execution tree and CCF creates a non-determinism node at the point the path constraint diverges. This helps to minimise the effects of non-determinism as they are localised to the smallest subtree that requires non-determinism. As non-determinism makes it impossible to guarantee correctness, we will often assume no non-determinism occurs when discussing the properties of the memory model.

## 3.5 Memory in Symbolic Execution

Writing a concrete value to a concrete address is trivial. Writing a symbolic value to a concrete address is also simple, we write the symbolic expression to the address specified. We can treat concrete and symbolic values in the same way by storing concrete values as symbolic expressions with a concrete value.
Writing to a symbolic address is more complicated. The write could affect different locations for different input values. We could create an if expression at each possible location, converting a write to

$$array_{next}[index] = if(\lambda = index)\, then\, \lambda\, else\, array_{prev}[index]$$

for each possible index. If we did this for every write then the expressions would become very complex and this may impact the solver performance.
We can avoid these drawbacks by using an update list, which maintains a record of the original array and stores modifications to the array in a list. This means a write to a symbolic address only requires a single entry, instead of N entries as in an array. For this reason, many popular symbolic executors use this technique [5, 6]. Unfortunately, CCF does not currently support this feature.

# Chapter 4

# Related Work

As each programming language can have a unique memory model, a variety of memory models have been developed. However, as C is the language used for the operating system, all languages must understand the C memory model to be able to communicate with the operating system. The C memory model is also used for Foreign Function Interfaces, which means a programming language must understand C to be able to call functions written in other languages. As a result, the C memory model is highly influential.

## 4.1 C Memory Models

C only has one official memory model, which is described fully in the C standard [14]. However, this description leaves some aspects to be defined by the implementation, so it would be reasonable to talk about the memory model used by GCC [11], Clang [18], and other implementations.

### 4.1.1 Name binding

Name binding is a simple form of memory management where memory objects are allocated and associated with a name in the program. When an object is created we give it a name that is not used for any other live object. By referring to the name of an object, this should uniquely identify the object allowing us to model objects as completely separate from each other. This model is intuitive and allows us to make strong claims about how memory is accessed in programming languages where it can be applied. This model is simple and easy to implement, however, C requires objects to be resident in memory and gives ways to access the memory representation of each object.

The name binding model can be extended to support references by tracking which names refer to which object. This makes the model suitable for use with many high level languages that do not allow the use of pointer arithmetic and would make it possible to make strong claims about which objects it is possible for any variable to refer to. Pointer arithmetic means that memory objects must have positions relative to each other, meaning the name binding model is insufficient to model C with pointers [28].

### 4.1.2 Array Model

Modelling memory as an array is used in many applications [5, 6, 19, 23]. When memory objects are allocated they inhabit a region of the array and valid memory accesses must lie within one of these regions. This makes an array model more appropriate for modelling memory for a C program as C guarantees that an object will be resident in memory for the duration of its lifetime and any accesses to the memory occupied by the object must be valid for that duration.
One way to implement an array model is to directly map memory into a single array, this allows for very fast access and write times as these operations can be implemented directly as array operations. One application that uses a direct mapped array is ASAN [24]. The disadvantage of using a single array is that it uses a large amount of memory as each byte of program memory needs to be represented by at least one bit in the array. If the program does not use the entire address space this could be very wasteful and if the memory model requires more metadata per byte of memory a flat array can quickly become infeasible. Therefore any implementation of an array memory model must decide which portions of memory to represent and how much metadata is stored per byte.
EXE [6] and KLEE [5] represent memory allocations using one array per memory object. This offers a clear advantage over using a single array as we only allocate memory that we will use. However, it also introduces complications if a memory access may span over multiple arrays. For symbolic executors smaller arrays are necessary, as a write to a symbolic address in an array can become problematic if the array is too large. Using an update list only requires one record to be stored per update, which is more efficient than recording every possible value for the state of the array. A limitation of this model is that pointers to pointers must be concretised to access the second level pointer, however this isn't an issue for concolic execution.
KLEE [5] represents memory similarly to EXE [6]. It differs by sharing immutable state between program executions to allow copy-on-write. This allows for faster branch execution as the heap can be copied in constant time.
Unfortunately, CCF does not support the theory of arrays used by KLEE to efficiently describe array interactions for the SMT solver to reason about. As a result, if we use an array model in CCF we will need to find an alternative way to allow exploration based on accesses to specific sub-objects contained within the memory objects that we can model using the theory of quantifier-free bitvectors.

### 4.1.3 Provenance Model

The C standard is unclear on how a pointer may be constructed to an object [20] and Defect Report 260 [26] states that implementations can track the origins of pointer values. This imposes a restriction not expressible in the array model as it requires that pointers cannot be constructed to objects arbitrarily, even if the program is able to guess the address of an object. A provenance model prevents this by maintaining

a record of objects which a given pointer has knowledge of and only allows pointers to access objects within their provenance set [20]. This limits the objects that a given pointer may access which allows more effective analysis and compiler optimisation and also detects more bugs which could be hidden by the array model, such as if a buffer overflow overran an adjacent object.
However, a provenance model is significantly more complex than an array model and it can produce unintuitive behaviour. It has been shown that many expert C programmers fail to correctly identify how provenance semantics will affect the behaviour of C programs [20]. As a result, a provenance model could report many errors that programmers would view as false positives, even if they are real bugs with respect to the provenance model.

### 4.1.4 Region Model

Region models have been developed for use with static analyses [9, 29]. This model represents every lvalue as a region of memory. Unlike the array model, this can allow regions to be created hierarchically. Static analysis of memory models has often struggled to handle pointer conversions as this makes types in C weak. This model handles pointer conversions by representing the result of the cast as a view on the region. Modifying a view then invalidates other views, guaranteeing that no undefined behaviour can be triggered by writing to multiple views. Although this model is useful for static analysis, in concolic execution we know the exact locations of the objects and this means we can detect memory bugs directly.

## 4.2 Static Analysis

In an ideal world the compiler would find all problems with any program immediately, so there could be a guarantee that if the program runs then it runs correctly. Unfortunately, static analysis is undecidable [17], which means we cannot determine non-trivial properties with certainty for all programs. As a result, static analysis is inherently limited in its ability to find problems with programs.
Despite this, there has been a lot of effort put into producing effective static analyses [13, 4, 27]. A static analysis is an algorithm that determines a property about a program, without executing the program. These analyses are used frequently in compilers for code generation [27] and for error detection [4]. The advantage of determining properties statically is that, providing the analysis is correct, a statically determined result must hold for any execution. This can allow a compiler to make optimisations confidently, knowing that it cannot break program behaviour. However, this confidence comes with the drawback of undecidability, so static analyses are forced to make a trade off between guaranteed correctness or guaranteed termination.
Static analysis is relevant to finding memory errors, as analyses that track pointer values[27] can allow us to determine which values it is possible for a pointer to reference. This can identify invalid memory accesses if we can determine that a pointer will reference an invalid object. In practice however, these analyses often

fail to determine if an access is invalid as a pointer could take a large number of possible values. A dynamic approach, which executes the program and determines if invalid operations occur in that execution can be more effective as it has access to more information that can only be determined at run-time.

## 4.3 Symbolic Execution

### 4.3.1 KLEE

KLEE [5] is a popular symbolic execution tool, which has been extended to support concolic execution. KLEE interprets LLVM IR, which allows it to perform symbolic execution on C programs that have been compiled by the Clang compiler. LLVM IR is simpler to interpret than working with C programs directly and is multi-platform unlike interpreting assembly. However, the misalignment between wanting to find bugs in C programs and searching for bugs in LLVM IR can potentially cause issues. One example of this is that Clang makes assumptions about the correctness of the input program, and can change program behaviour even with all optimisations turned off [25].
KLEE's object memory model is an instance of an array memory model, where each object is allocated its own array. As a symbolic executor, KLEE focuses on efficient process duplication, which is less important for a concolic executor. KLEE's implementation of symbolic arrays uses an update list which allows for efficient storage of symbolic modifications of arrays. As memory is modelled using a collection of arrays this is important to produce efficient SMT queries.

### 4.3.2 SymCC

SymCC [23] is a recent concolic executor which aims to achieve greater performance through executing compiled code directly. As the code is not interpreted, it must be instrumented at compile time to insert code to track memory allocations. They implement a shadow memory to record symbolic data related to the execution.

## 4.4 Compiler Sanitizers

GCC and Clang implement instrumentation tools called sanitizers which insert run-time checks into the compiled binary to determine if a bug is present at execution time. This can be useful in cases where static analysis fails to identify an error, but it also fails to eliminate the possibility of a bug. For many types of undefined behaviour it is significantly easier to identify a bug at execution time when we only deal with concrete values and a single execution path. For these issues, a sanitizer can identify where the issue happens by adding extra checks into the program and halting execution if the program enters an illegal state.
Sanitizers are not used in production software as they result in a large slowdown due to the extra safety checks at run-time. Instead, they are typically used in a test

suite to check that the code under test does not exhibit illegal behaviour. This means sanitizers are subject to the same limitations that manual testing has, namely that it can only demonstrate a bug if we provide an input that causes the program to follow a specific execution path that leads to the illegal behaviour. It is possible to have a large test suite and still miss these cases, so the sanitizer would not find any bugs.
The AddressSanitizer [24] (ASAN) built into Clang and GCC adds instrumentation to report memory bugs at run-time. To find these bugs ASAN maintains a shadow memory, a direct mapping that encodes each 8 byte segment of memory into a single byte of shadow memory. When an address is accessed, ASAN will check if the corresponding shadow memory is in a valid state to access. To do this it has to monitor all memory allocations and frees and update the shadow memory to mirror these changes. It also poisons the surrounding memory, to make it easier to determine if the program under test reads invalid memory. CCF uses a similar shadow memory mechanism, but instead of directly mapping memory, it maintains a binary tree of memory objects. This allows CCF to store information dynamically per memory object, which is important for a concolic executor that may need to maintain a lot of state for some memory regions and almost no data for regions that are never accessed.

# Chapter 5

# Design

To find memory bugs we need to define the memory model we are using, design a checker to detect memory bugs during execution, and then produce constraints to allow us to find new inputs.
CCF has an existing framework implemented to handle memory interactions. The C compiler is modified to insert a function call on each memory access, providing both the concrete data and symbolic expressions required to fully describe the memory access. The concrete information is handled by the framework to allocate, deallocate, read, and write memory. The framework provides a dummy memory model implementation which accepts all memory accesses as valid.
We will implement a memory model to define how memory can be accessed by the system under test. The implementation of this model should replace the dummy memory model and be able to validate and construct constraints on the memory accesses performed by the system under test.

## 5.1 Memory Model

The memory model defines how a program may interact with memory. The main memory operations the memory model must support are allocations, deallocations, reads, and writes. Together these describe all the ways that the program can affect memory. We can treat reads and writes in the same way so we call this an access.

$$MemoryOp :: (\mathit{AddressSpace}, \mathit{LiveObjects}, \mathit{AccessRange}) \to \mathit{LiveObjects}$$
$$MemoryOp ::= \mathit{Allocate} \mid \mathit{Deallocate} \mid \mathit{Access}$$

Memory is composed of a set of bytes. We are only concerned with how the program interacts with memory, which means we can abstract away most of the architectural features of modern memory hardware. We can treat all bytes of memory the same, regardless of whether memory is segmented or how virtual memory is implemented. This makes the memory model applicable across different architectures. We require that all accesses lie within the address space. As we are using x86_64, memory allocations will be contiguous which simplifies how we can express memory operations.

An allocation creates a new memory object. This is represented as reserving a range of addresses for the object. For an allocation at position $pos$ with size $n$ we need to set $[pos, pos + n)$ to be allocated. As we don't need to store per byte information, we can equivalently create an object representing this allocation that stores the position and size of the memory object. To validate an allocation, we need to ensure that the range $[pos, pos + n)$ is currently unallocated as we do not allow memory objects to overlap.

$$\forall x \in o.\, x \cap r = \emptyset \iff Allocate(\alpha, o, r) = o \cup \{r\}$$

A deallocation removes a memory object from the set of live objects, freeing the range of addresses reserved for the object. This operation is the inverse of an allocation, and sets a range $[pos, pos + n)$ to be deallocated. To validate a deallocation we ensure that the range $[pos, pos + n)$ is allocated. Unlike memory allocations, C programs are able to specify the address to be deallocated, introducing more possible errors. The address to be deallocated must be an address previously returned by a memory management function and the region must not have been deallocated by a previous call to free or realloc. In an memory model, we do not directly track which addresses have been returned by memory management functions. As the memory management functions return pointers to the start of objects, we can impose this requirement by ensuring the address to be deallocated is the start of a memory object.

$$r \in o \iff Deallocate(\alpha, o, r) = o - \{r\}$$

Reads and writes are treated equivalently in our memory model. We do not record information regarding which bytes have been written or read so the array does not need to be updated to track these modifications. We require that the region accessed is entirely contained within a single allocation.

$$\exists x \in o.\, r \subseteq x \iff Access(\alpha, o, r) = o$$

C-objects also have alignment information which defines which addresses they can be read from or written to. We do not represent alignment and cannot catch memory bugs caused by incorrect alignment.

### 5.1.1 Constraints

Once a memory access has been validated by the memory model during execution, we should produce a constraint on the symbolic input to describe the program path. This will allow the SMT solver to produce new inputs that access other memory objects, if such inputs exist.
The sequence of constraints produced during a program execution should uniquely identify one path through the program. We can consider each memory access to be a branch and the set of branch targets is equal to the set of memory objects that the memory access could hit plus an option to have accessed memory not contained within any memory object if this may have occurred.

$$\begin{tikzcd}
 & \text{cond} \arrow[dl] \arrow[d] \arrow[dr] & & & & \text{cond} = 1 \arrow[dl] \arrow[dr] & \\
1 & 2 & 3 & & \text{cond} = 2 \arrow[dl] \arrow[dr] & & 1 \\
 & & & 3 & & 2 &
\end{tikzcd}$$

**Figure 5.1:** A branch with $k$ targets can be converted to $k-1$ binary branches.

When a memory access occurs under concolic execution and the access hits an object, we need a way to identify that object to discriminate against accesses to different objects. We cannot use the address of the object, as this is non-deterministic. Instead, we allocate a unique id value to each object based on the allocation order. Assuming no non-determinism has occurred, whenever the program follows the same execution path the same number of objects will be allocated. This guarantees that objects with the same id across executions are in some sense the same object, even if they exist at different addresses.

$$\frac{m \leq \lambda \leq \lambda + k \leq n \wedge \{m \dots n\} = \{i \,|\, \alpha_i = Alloc(id)\}}{generate\_constraint\; Access(\alpha, \lambda \dots \lambda + k)\; id\, m\, n \rightarrow \{m \leq \lambda \wedge \lambda + k \leq n\}}$$

When a memory access hits a memory object, we produce a constraint that ensures the access is entirely contained within that memory object. If no object is accessed, we have found a memory bug. However, we still need to produce a constraint to guide the program exploration to find other possible bugs in the program. As we have hit a case where no object is accessed, the constraint can be described as the negation of an access to any object. If we represent an access to a memory object with id: $id$ as $hit(id)$, this means not hitting an object takes the form $\neg \bigvee(hit(id))$.

## 5.2 Handling Branches With Many Targets

A key difference between symbolic and concolic execution occurs when the program reaches a symbolic branch with multiple targets. In symbolic execution we query the SMT solver for which branch targets are possible and fork over all possible targets. In concolic execution we maintain a separation between program execution and exploration and we don't want to pause execution to query an SMT solver as this would slow down execution and break this separation. As a result, when we reach a branch we do not know all the possible targets.

CCF handles this by recording the constraint required to take the specific branch followed by the concrete execution at each branch point. Once a program finishes execution (successfully or via a crash) the sequence of constraints generated (called the path constraint of the execution) is added to the execution tree managed by CCF. CCF does not make the execution tree available to the process under test to enforce separation, but this means during execution we do not know which branches have already been explored.

The majority of branches in a program are binary branches and a non-binary branch can be converted to a series of binary choices by branching on each possible case in order. Figure 5.1 shows how CCF performs this binarisation to convert a branch with $k$ targets to $k$ binary branches. CCF takes advantage of this to represent all branches as binary branches which makes the execution tree a binary tree. This means during exploration we always know all possible cases as we either take a branch or we don't. However, when we introduce a memory model and start to branch over all possible targets of each memory access, the average number of branch targets will increase and the previously uncommon case of large switch statements with many targets becomes a common scenario that is important to encode efficiently.

We modify the execution tree to treat all branches as having k possible targets. This simplifies the model of the execution tree as it means one node in execution tree represents one symbolic branch in the program. This should help to make it easier to reason about the exploration behaviour of CCF.

An alternative approach we considered was replacing the linear search with a binary search. CCF searches for the correct case from the start in a linear search which adds on average $\frac{n}{2}$ nodes. As both memory addresses and switch condition variables are well-ordered, we could use a binary search to find the correct case which would improve the time taken to $\log(n)$. However, implementing choice-out-of-k nodes would allow us to represent a choice with only 1 node, minimising the size of the execution tree.

# Chapter 6

# Implementation

We implement a memory model into CCF and adapt CCF's internal representation of the execution tree to better represent memory accesses.
CCF has an existing framework that updates a store of metadata regarding the symbolic state associated with the program execution. This framework supports memory accesses by providing a memory model interface. To implement a memory model for CCF we need to implement a memory model checker and then extend the behaviour to generate constraints.

## 6.1 Checking the Memory Model

The memory model represents memory objects as occupying a contiguous region of a byte array, where the start of the occupied region for an object $x$ is the offset $\&x$ and the size of the occupied region is $sizeof(x)$. To implement such an array directly would have an immediate issue, the size of the byte array would need to be equal to the total addressable memory. For a 64 bit architecture this could be up to $2^64$ bytes or 16 exibytes, which would require almost as much RAM as google chrome.
CCF's memory framework uses a binary tree to reduce the space required by only storing information about allocated objects. Binary look-up trees allow us to look up addresses in $\mathcal{O}(\log n)$. For each memory object we store the size of the allocation and a unique id and an access to any other region of memory will access an Unallocated element. This greatly reduces the amount of space required to model memory.

$$MemoryObject ::= Unallocated \,|\, Allocated(id, size)$$

$$\forall x \in o.\, x \cap r = \emptyset \iff Allocate(\alpha, o, r) = o \cup \{r\} \tag{6.1}$$

To implement allocations according to (6.1), we need to ensure that the entire range we allocate is unallocated. We can do this directly by iterating over memory objects in the binary tree that intersect the range and checking that they are all Unallocated. We ensure that we allocate a unique id to each memory allocation by using an atomic counter. Although CCF does not currently support multi-threaded programs, we ensure that the code is thread-safe so that CCF can be easily extended in the future.

$$r \in o \iff Deallocate(\alpha, o, r) = o - \{r\} \tag{6.2}$$

(6.2) describes how we model deallocations. To validate a deallocation we check that the range to be deallocated exactly coincides with an allocated memory object in the binary tree.

$$\exists x \in o.\, r \subseteq x \iff Access(\alpha, o, r) = o \tag{6.3}$$

Reads and writes are described by (6.3). We check that the range of the access is contained entirely by a single allocated memory object. For all of these checks we are working with the concrete range that is being accessed, not the symbolic expressions associated. This is because during the execution of a program we only want to report an error that actually happens. This guarantees that all errors reported are real, avoiding false positives.

### 6.1.1 Finding new inputs

To find new inputs we need to generate constraints from the symbolic state we store in the concolic runtime. CCF updates the symbolic state for each operation that occurs in the system under test (SUT). This includes constructing the symbolic expressions that represent the symbolic addresses used for memory accesses. We implement constraints using these symbolic expressions.
The simplest constraint we could generate for an access to address $sym\_ptr$ with size $sym\_size$ which hit a memory object starting at $obj\_pos$ with size $obj\_size$ would be $obj\_pos \leq sym\_ptr \wedge sym\_ptr + sym\_size \leq obj\_pos + obj\_size$. This works well for a basic implementation of the memory model.
CCF has a framework for building constraints based on LLVM IR. This means that the we can easily create symbolic expressions that match the LLVM instructions that have occurred. To represent the previous expression using CCF's framework we have:

$$\begin{aligned} &Binary(And, None, Binary(ICmp, \textbf{ICMP\_ULE}, obj\_pos, sym\_ptr),\\ &\qquad Binary(ICmp, \textbf{ICMP\_ULE}, Binary(Add, None, sym\_ptr, sym\_size),\\ &\qquad\qquad Binary(Add, None, obj\_pos, obj\_size))) \end{aligned}$$

A better way to represent memory accesses would be using the SMT theory of arrays. As CCF is a prototype, CCF's constraint framework does not support the theory of arrays yet. If CCF supported the theory of arrays it would be possible to represent a memory access as a single array access, instead of a series of bitvector operations.
CCF matches execution traces to the execution tree by walking down the tree from the root and checking that each constraint encountered in the execution tree matches a valid option at that point in the execution tree. It matches the constraints by directly comparing them and checking if the constraint equals the constraint or its negation. Unfortunately, memory allocation addresses are inherently non-deterministic. This means that checking against the constraints described above would never result in a match as the $obj\_pos$ is non-deterministic.

To overcome this, we changed the execution tree to match constraints using a unique id for each code location in the LLVM IR. As LLVM IR does not store code location information by default, we modified CCF's instrumentation to generate a 64 bit id for each code location using a pseudo-random number generator seeded with the path to the source code file. This guarantees that the id generated for a specific code location will always be the same between runs. Once we have a unique id for a code location, we can match branches on non-deterministic data like memory addresses by checking that the code locations match. This comparison is also more efficient than comparing expressions directly, as it takes constant time to compare the equality of two 64 bit values.

### 6.1.2 Sub-Objects

C-objects can constructed recursively, allowing us to create complex data structures such as structs that contain structs. Although this gives us a lot of freedom to change how data is arranged, it is difficult to represent in the memory model. This is because memory objects do not have internal structure and cannot be defined recursively, they only have a position and a size.

As some memory bugs depend on accessing specific sub-objects, it would be impossible for the memory model to find these bugs. To illustrate the problem, consider a double indirection by way of an array of pointers as in 6.1. Under concolic execution, a direct implementation of the memory model would produce a constraint with two cases, either the memory access $arr[inp[0]]$ is within the memory object pointed to by $arr$, or it is not. If we got unlucky and failed to try the input $0x2$ immediately, the case where the memory access was within the memory object would be considered satisfied and we would not try to access other sub-objects of $arr$. As a result, we would not detect the null-pointer dereference.

The best way to handle this would be to model the contents of the memory object using the theory of arrays. This would allow us to construct a symbolic value representing an access to any portion of the memory object. Unfortunately, CCF does not currently support this SMT theory.

To work around not having access to the theory of arrays, we allocated ids $[id \ldots id + size)$ to each memory object, allowing us to treat an access starting with each byte of an object as a separate case. This has the immediate downside of exhaustively searching each accessible byte of each memory object, even if this has no further effect on the execution. This results in $\mathcal{O}(n)$ SMT queries per object instead of $\mathcal{O}(1)$, but as this allows us to detect many memory model violations that would not otherwise be found, this is an acceptable trade-off.

To allow for accessing sub-objects, we introduce a per byte object id $obj\_id = obj\_pos + offset$, and create the constraint $sym\_ptr = obj\_id \wedge sym\_ptr + sym\_size \leq obj\_pos + obj\_size$. This means that we exhaustively search for all possible byte accesses allowing us to find more bugs. If CCF supported the SMT theory of arrays we would be able to do this much more efficiently, but this trade off still allows us to find bugs in programs that we would not be able to if we only considered accesses to whole memory objects.

```
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>
#include <stdlib.h>

int main(void) {
    u_int8_t inp[6];
    int n = 4;
    int** arr = calloc(n, sizeof(int*));
    arr[0] = calloc(1, sizeof(int));
    arr[1] = calloc(1, sizeof(int));
    arr[3] = calloc(1, sizeof(int));
    int bytes_read = read(STDIN_FILENO, &inp, 1);
    if (inp[0] > 3) {
        return 0;
    }
    int x = arr[inp[0]][0];
    return 0;
}
```

**Figure 6.1:** This program performs a double indirection by way of an array of pointers. Without considering sub-objects, we would not detect the error in the program.

### 6.1.3 Limitations

C-objects have an associated memory alignment which restricts which addresses they can be written to and from. The memory model does not represent this as it treats each memory object as having no internal structure. Therefore, we cannot detect unaligned accesses even though these can cause crashes on some hardware.
The implementation of sub-objects has bad scaling for large memory objects with many accessible bytes.

## 6.2 Choice-Out-Of-K

CCF's original implementation of the execution tree only allowed for binary choices which was reasonable as most branches in a program are binary. In order to encode a switch statement using only binary choices and allowing for matching against previous traces, the original implementation would emit a binary constraint of $\neg(condition\_expr = case)$ for each previous switch case that we did not hit and finally emit a binary constraint of $(condition\_expr = case)$ for the case we hit. This meant we emitted on average $\frac{n}{2}$ constraints for each switch we encountered. This was acceptable for switches as in real world cases we rarely see extremely large switch statements. However, the number of valid targets for a memory access under the memory model is equal to the number of memory objects allocated at that

execution point. This means every memory access behaves like a large switch.
A natural extension of the execution tree would be to allow each node in the tree to have $k$ children, which would allow each switch and memory access to represented using a single node in the execution tree. Not only does this reduce the number of constraints we need to emit per branch to a constant (1), but it also means the execution tree generated more closely matches the definition of the execution tree of having each node represent a branch.
One way to implement choice-out-of-k nodes is to represent all branches as switches. A switch statement is defined by a condition expression, a set of cases which each have a constant case value, and a default case which handles all other values. This can generalise binary decisions by using the binary condition expression as the switch condition expression and having one case with a case value of true and use the default case for when the condition expression evaluates to false. It can also generalise memory accesses using the address expression as the condition expression and each memory object is a separate case. This leaves the case where no memory object was accessed as the default case. The main advantage of this approach is reducing expression duplication. As we only store the condition expression once we limit the memory used and we can generate the full constraint to pass to the solver only when it is required as we store all the necessary information in the execution tree.
When creating a choice-out-of-k node, we have to choose between transmitting all possible cases immediately or transmitting cases only when we reach that case in a concrete execution. Transmitting cases lazily has a clear advantage as it means we never transmit an unsatisfiable case, reducing the number of SMT queries we have to execute to find all possible cases. However, when we reach a default case we then need to transmit the negation of an over-approximation of the possible cases to ensure we still allow all possible cases to be found. This should minimise the number of queries but means that the default case requires us to duplicate the information of all other cases which could waste memory.
To distinguish choices, we record a unique id per branch, memory access, and switch, the condition expression, and the case. We need to compare both the id and the condition expression as the expression not change for different cases, so we need to insert a non-determinism node if the expression is not equal even if the ids are equal.

# Chapter 7

# Evaluation

To evaluate the contribution of this project, we need to evaluate the application of our memory model to the concolic execution of C programs, our implementation of the memory model, and the implementation of the choice-out-of-k abstraction for nodes in the execution tree.

The most important feature of the implementation of the memory model that we need to verify is its correctness. To evaluate correctness, we have created a suite of test programs that interact with memory in various ways. A correct implementation should report memory model violations in each test that violates the memory model and should avoid reporting violations for all that do not. These test programs do not depend on any external input, which will allow us to evaluate the correctness of the array memory model implementation independent of the concolic program exploration. As CCF did not previously have a memory model implementation, it will not be possible to comparatively evaluate the performance of the memory model implementation. However, we can still collect performance metrics for the test suite to demonstrate that the memory model implementation is reasonable.

In practice, it is unlikely that a program which may violate the memory model would do so without any external input. To evaluate how the memory model implementation facilitates program exploration during concolic execution to find possible inputs that violate the array memory model, we have a suite of test programs that consume external input and vary their memory interactions depending on that input. A successful implementation should be able to find inputs that violate the array memory model in programs where such inputs exist, and should never erroneously report memory model violations. A key advantage of concolic execution over static analysis techniques is that it never produces false positives, so it would be a shame to lose this property due to an overly restrictive memory model.

Concolic execution is an inherently inefficient process, which can have poor scaling in two main ways. The first is the use of an SMT solver which can have exponential run-time in the worst case. To evaluate the performance of the SMT queries that are produced when running CCF with the array memory model we can compare the average time taken per SMT query between CCF with and without the array memory model across a variety of test programs as well as the total number of SMT queries produced. A successful implementation should avoid degrading the performance of the SMT solver. The other way concolic execution scales poorly is

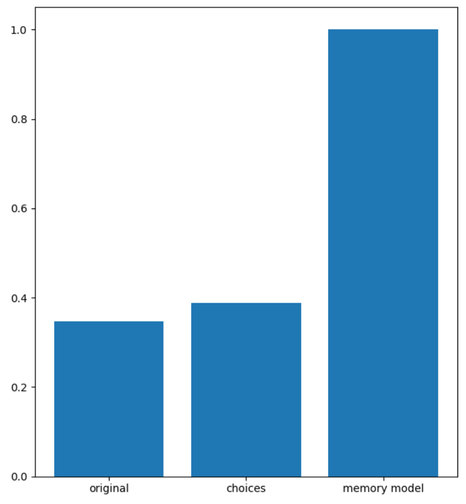


**Figure 7.1:** The proportion of successful tests out of the 49 synthetic tests created. Tested with the original version of CCF, CCF with the choice-out-of-k nodes, and CCF with choice-out-of-k and the memory model. The memory model clearly results in an increase in the ability to detect memory bugs.

in the exponential blow-up in the number of paths. This means it is important to minimise the number of nodes to be explored as much as possible without blocking the exporation of any possible path. Collecting the number of nodes, as well as other relevant information such as the number of SAT and UNSAT nodes, will help in evaluating how the memory model and choice-out-of-k implementations have affected the performance of the program search.

Evaluating the choice-out-of-k implementation is more straightforward as we can compare the implementation against the exclusively binary implementation that is already implemented in CCF. Running test programs with both versions will allow us to directly compare the execution time as well as other relevant statistics. As CCF does not have a memory model (other than the array memory model implementation described here), these tests will need to avoid symbolic memory accesses to allow a fair comparison between each implementation.

Figure 7.1 shows the results of executing the synthetic test suite with the original version of CCF, modified to include some bug fixes, CCF with choice-out-of-k nodes implemented but using a behaviour-less dummy memory model, and CCF with choice-out-of-k nodes and the array memory model implementation. We can see that the original version of CCF passes 17 out of 49 tests, corresponding to the tests that ensure no error is reported for correct program behaviour. This is the expected result as CCF does not use a memory model implementation so it should not produce any errors. With the implementation of the array memory model, the ability to detect memory bugs is greatly increased and CCF is able to correctly produce error messages for all tests that contain errors. This shows that the implementation of the array memory model matches the design of the array memory model described

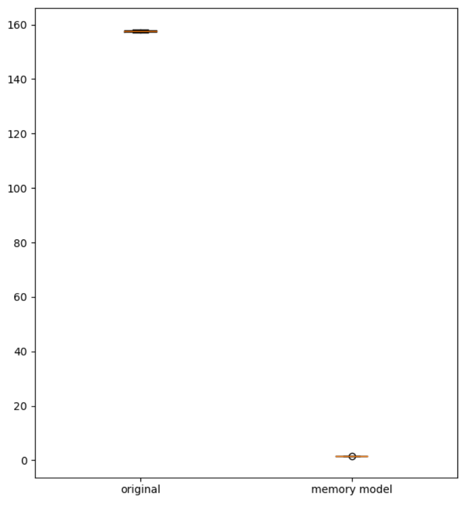


**Figure 7.2:** The time taken to compile a program containing a large switch statement using the original version of CCF and CCF with the array memory model and choice-out-of-k nodes. Each box-plot shows the compile times from a sample of 30 compilations.

earlier.

Figure 7.2 shows how the implementation of choice-out-of-k nodes makes compiling large switch statements significantly faster. We would expect this as the previous algorithm to convert switches into binary choices emitted $\mathcal{O}(n^2)$ binary constraints in total while using choices only requires us to emit $\mathcal{O}(n)$ constraints. As well as giving a large speed-up this also means that programs that use significantly larger switches can now be compiled, as in testing we found that clang would fail to instrument very large switch statements with the original version of CCF.

Figures 7.3 and 7.4 show the performance of running CCF on a simple program that loops until the loop counter equals a symbolic variable. We can see that adding the array memory model introduces a statistically significant overhead but the overall performance is comparable.

Figure 7.3 shows the number of nodes in the execution tree over time. We see that the original version of CCF creates all the nodes immediately, while the versions with choice-out-of-k add the nodes gradually over time. This doesn't have a significant impact on the performance of the program, but for larger programs could save resources by only adding nodes when needed. Importantly, the choice-out-of-k versions never add unsatisfiable nodes which could waste a significant amount of memory in the original version.

Figure 7.4 shows the number of nodes explored over time. We see that the original version and the version with choices but without the array memory model have similar performance, while adding the array memory model is slower. This is expected as adding the array memory model requires the program to do more work in creating extra constraints and solving more complex SMT queries. The test program only uses binary branches, so it is good to see that the performance has not significantly

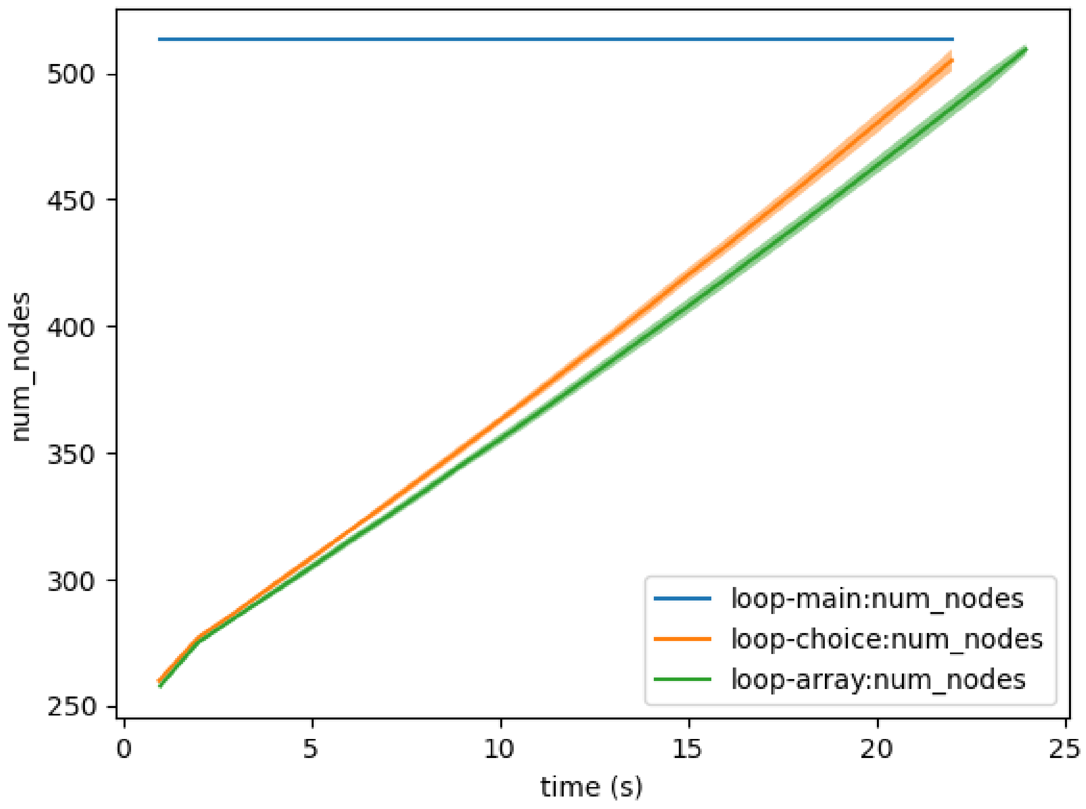


**Figure 7.3:** The mean number of nodes generated over the runtime of a simple loop with a confidence interval of 99.7%. Each test was run 30 times.

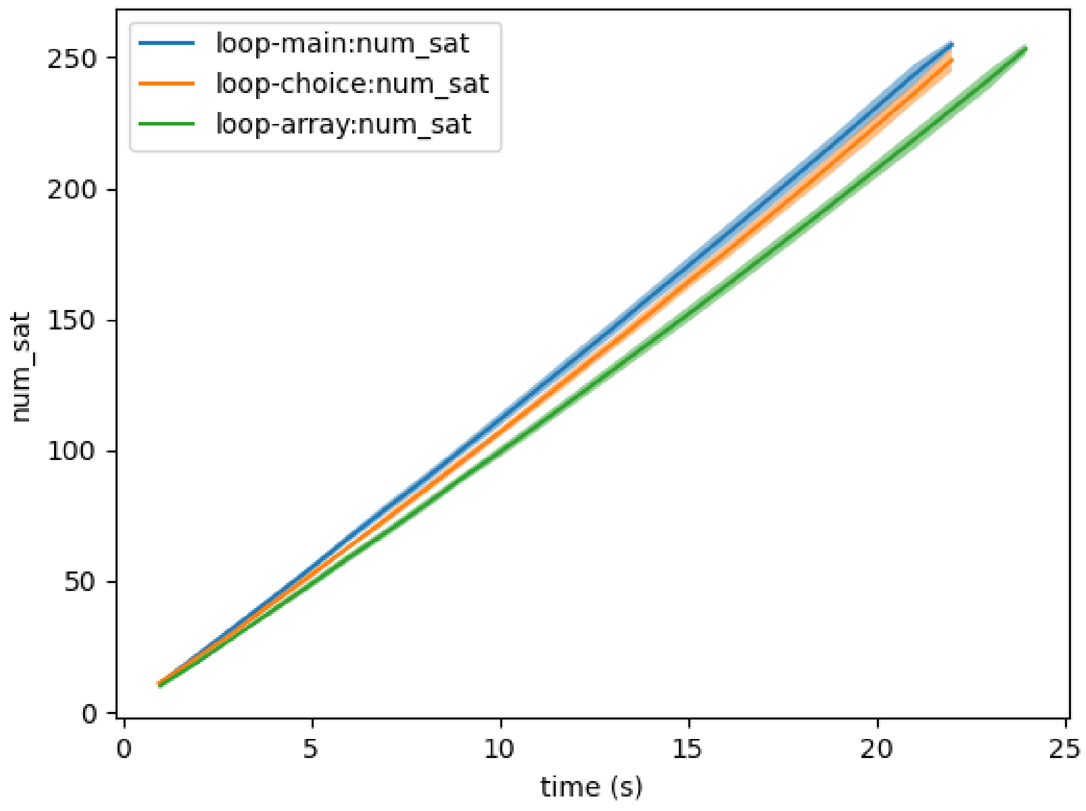


**Figure 7.4:** The mean number of satisfied nodes generated over the runtime of a simple loop with a confidence interval of 99.7%. Each test was run 30 times.

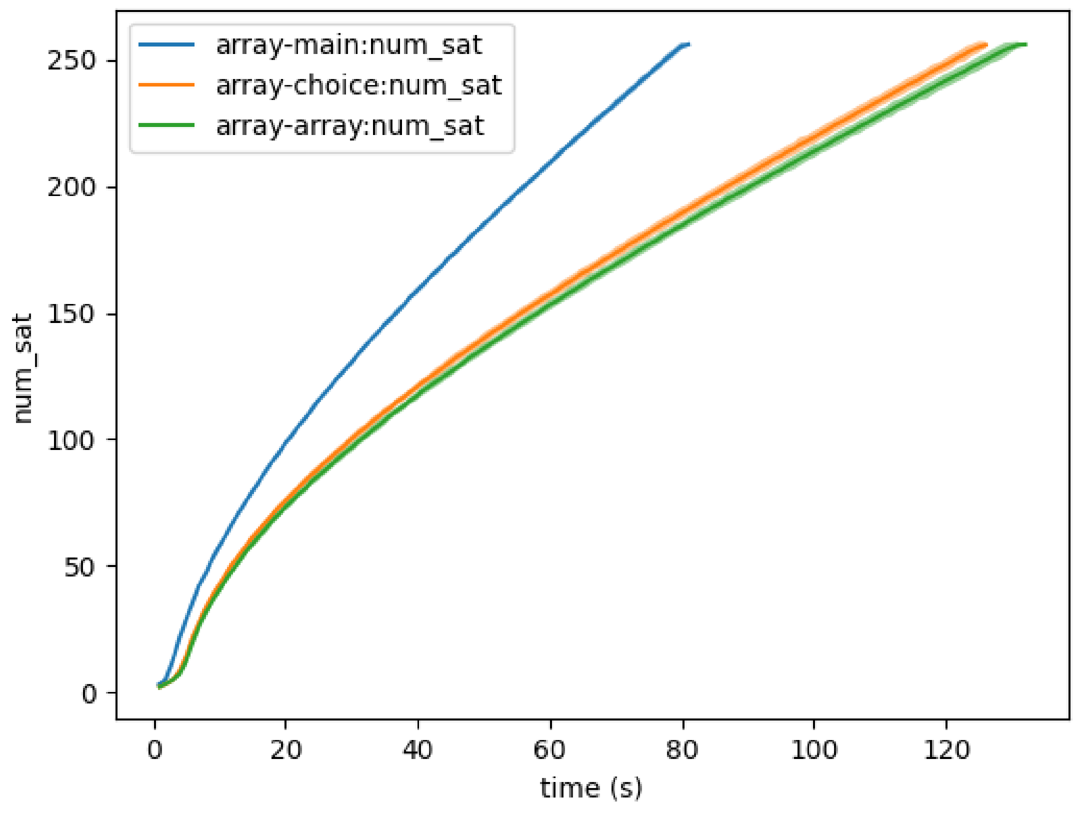


**Figure 7.5:** The mean number of satisfied nodes generated over the runtime of a loop that performs a memory access to a symbolic address, with a confidence interval of 99.7%. Each test was run 30 times.

degraded when using choice-out-of-k nodes for the binary case. As the binary case is the most common type of branch, it is important that we maintain good performance for this case and the fact that it is possible to generalise branches to allow any number of nodes without making this case perform worse justifies the implementation of choices.

Figure 7.5 shows the number of nodes explored for a different program. This program repeatedly makes accesses to a symbolic address. We would expect the array model (in green) to take significantly longer than the original implementation (in blue), as the original version does not model memory so the version with the memory model implemented will do more work. It is surprising to see that the version with choices but using the dummy memory model has almost the same performance as the version with the memory model. This means that either the implementation of choices is inefficient or the overhead of transmitting memory constraints greatly slows down the program.

### 7.0.1 Threats to Validity

One concern may be that the implementation of the array memory model may not match the design of the memory model. As the synthetic test suite tests the behaviour of each core function of the array memory model implementation and we pass all the synthetic tests we are confident that the implementation matches the design.

# Chapter 8

# Conclusion

We have implemented a memory model using CCF's memory framework and implemented a system to generate constraints on the memory accesses to guide concolic execution to find new paths. We see that after implementing the memory model CCF is able to detect violations of the memory model in a variety of test programs. This should make CCF a more effective tool for finding bugs in C programs.

## 8.1 Future Work

This work is limited by the lack of support for the theory of arrays in CCF. Once arrays are supported the memory model could be extended to make general queries about the contents of the array. This would improve efficiency over the current implementation by reducing the number of queries executed. It is also possible to implement different memory models such as a provenance aware model which could detect additional memory bugs.

# Bibliography

[1] C. Barrett and C. Tinelli. *Satisfiability Modulo Theories*, pages 305–343. Springer International Publishing, Cham, 2018. pages 11

[2] R. S. Boyer, B. Elspas, and K. N. Levitt. Select—a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*, page 234–245, New York, NY, USA, 1975. Association for Computing Machinery. pages 10

[3] R. Brummayer and A. Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In S. Kowalewski and A. Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. pages 11

[4] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802, 2000. pages 16

[5] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008. pages 11, 12, 13, 15, 17

[6] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):1–38, 2008. pages 13, 15

[7] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC ’71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery. pages 11

[8] N. Eén and N. Sörensson. An extensible sat-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 502–518, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. pages 11

[9] H. Hampapuram, Y. Yang, and M. Das. Symbolic path simulation in path-sensitive dataflow analysis. volume 31, pages 52–58, 09 2005. pages 16

[10] W. Howden. Symbolic testing and the dissect symbolic evaluation system. *IEEE Transactions on Software Engineering*, SE-3(4):266–278, 1977. pages 10

[11] https://gcc.gnu.org/onlinedocs/gcc/Contributors.html. GCC, the GNU compiler collection. Technical report, Free Software Foundation, Inc., 1987. pages 14

[12] T. Hynninen, J. Kasurinen, A. Knutas, and O. Taipale. Software testing: Survey of the industry practices. In *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1449–1454, 2018. pages 7

[13] S. C. Johnson. *Lint, a C program checker*. Bell Telephone Laboratories Murray Hill, 1977. pages 16

[14] JTC1/SC22/WG14. ISO International Standard ISO/IEC 9899:2018 - Programming Language C. Technical report, International Organization for Standardization (ISO), 2018. pages 5, 14

[15] J. C. King. A new approach to program testing. In *Proceedings of the International Conference on Reliable Software*, page 228–233, New York, NY, USA, 1975. Association for Computing Machinery. pages 10

[16] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, jul 1976. pages 10

[17] W. Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, dec 1992. pages 16

[18] C. Lattner. Llvm: An infrastructure for multi-stage optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, 2002. pages 14

[19] X. Leroy and S. Blazy. Formal verification of a c-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41:1–31, 2008. pages 5, 15

[20] K. Memarian, V. B. F. Gomes, B. Davis, S. Kell, A. Richardson, R. N. M. Watson, and P. Sewell. Exploring c semantics and pointer provenance. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019. pages 15, 16

[21] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, dec 1990. pages 7

[22] NIST. National vulnerability database - statistics. Technical report, Technology, N.I.o.S.a, 2024. pages 2

[23] S. Poeplau and A. Francillon. Symbolic execution with SymCC: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*, pages 181–198, 2020. pages 15, 17

[24] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, Boston, MA, June 2012. USENIX Association. pages 2, 6, 15, 18

[25] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 260–275, 2013. pages 17

[26] WG14. Defect report 260. Technical report, 2004. pages 15

[27] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for c programs. *ACM Sigplan Notices*, 30(6):1–12, 1995. pages 16

[28] Z. Xu, T. Kremenek, and J. Zhang. A memory model for static analysis of c programs. In *Leveraging Applications of Formal Methods, Verification, and Validation: 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part I 4*, pages 535–548. Springer, 2010. pages 15

[29] Z. Xu, T. Kremenek, and J. Zhang. A memory model for static analysis of c programs. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, pages 535–548, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. pages 16

[30] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294, 2011. pages 7

[31] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. QSYM : A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 745–761, Baltimore, MD, Aug. 2018. USENIX Association. pages 10

[32] M. Zalewski. American fuzzy lop - whitepaper. Technical report, 2016. pages 7