Aiding global function pointer hijacking for post-CET binary exploitation

Tesi di Laurea Magistrale in Computer Science and Engineering - Ingegneria Informatica

Authors:
Alessandro Bertani
Marco Bonelli

Student IDs: 944456, 944655
Advisor: Prof. Mario Polino
Co-advisors: Lorenzo Binosi, Stefano Zanero
Academic Year: 2021-22
Abstract

Exploits for memory corruption vulnerabilities often use control-flow redirection as means of achieving arbitrary code execution. For this reason, current state-of-the-art defense proposals have control-flow integrity (CFI) preservation as their main objective. A recent example is Intel’s Control-flow Enforcement Technology (CET): a hardware-assisted CFI preservation mechanism implemented in modern Intel Core processors, which makes it significantly harder for a potential attacker to perform control-flow redirection of a vulnerable program. However, post-CET exploitation is still possible through function pointer hijacking under the right conditions. If an arbitrary write vulnerability is present in a program, global function pointers of loaded libraries could be overwritten to possibly achieve arbitrary code execution bypassing state-of-the-art CFI defenses such as CET.

We propose an approach for identifying global function pointers defined in C libraries, their call sites reachable from exported library functions, and how to reach them. The identification of such library functions is done through static source code analysis, and symbolic execution is then used to identify and solve constraints on function parameters and global variables needed to direct the control-flow of a program to such call sites, with focus on the evaluation of complex parameters such as structure pointers. We present Untangle, an open-source tool that implements and automates this approach. Untangle was tested on several open source C libraries, identifying a total of 57 unique global function pointers, reachable through 1488 different exported functions, finding and verifying the correctness of the constraints to be satisfied in order to reach global function pointer calls through 484 of them.

Keywords: binary exploitation, control-flow integrity, control-flow hijacking, static analysis, symbolic execution, exploitation automation
Gli exploit di vulnerabilità di corruzione della memoria vengono spesso attuati tramite redirezione del control-flow di un programma per ottenere esecuzione di codice arbitrario. Per questo motivo, l’attuale stato dell’arte in termini di difesa ha come scopo principale la preservazione dell’integrità del control-flow. Un esempio recente è un’estensione di sicurezza sviluppata da Intel per gli ultimi modelli di processori Intel Core, chiamata Control-flow Enforcement Technology (CET). Meccanismi di difesa come CET rendono significativamente più complesso alterare il control-flow di un programma vulnerabile. Tuttavia, l’utilizzo di exploit che sfruttano la sovrascrittura di puntatori a funzione è ancora possibile in determinate condizioni. Un programma che presenta vulnerabilità che permettono la scrittura arbitraria in memoria, renderebbe possibile la sovrascrittura di puntatori a funzione globali definiti nelle librerie utilizzate dal programma, permettendo di aggirare meccanismi di difesa all’avanguardia come CET.

Con il nostro lavoro proponiamo un approccio per identificare puntatori a funzione globali definiti in librerie C, le loro chiamate raggiungibili da funzioni di libreria esportate, ed il modo in cui raggiungerle. L’identificazione di tali funzioni di libreria è effettuata tramite analisi statica del codice sorgente. L’esecuzione simbolica viene successivamente impiegata per trovare e soddisfare eventuali vincoli su parametri di funzione e variabili globali necessari per raggiungere chiamate di puntatori a funzione globali, con particolare interesse verso parametri complessi come puntatori a strutture. Abbiamo sviluppato Untangle, uno strumento open source che implementa ed automatizza questo approccio. Untangle è stato testato su diverse librerie C open source, nelle quali abbiamo identificato un totale di 57 differenti puntatori a funzione globali, le cui chiamate sono raggiungibili tramite 1488 funzioni esportate. Abbiamo infine identificato i vincoli necessari per il raggiungimento di tali chiamate e verificato la loro correttezza in 484 dei casi.

Parole chiave: binary exploitation, control-flow integrity, control-flow hijacking, analisi statica, esecuzione simbolica, automazione di exploit
## Contents

Abstract i
Abstract in lingua italiana iii
Contents v

1 Introduction 1

2 Background and Motivation 5
  2.1 Background 5
    2.1.1 Exploitation Techniques and Defenses 5
    2.1.2 Advanced Exploitation Techniques and Defenses 7
    2.1.3 Function Pointer Hijacking 10
    2.1.4 Static Source Code Analysis 10
    2.1.5 Symbolic Execution 11
  2.2 State of the Art 12
  2.3 Problem Statement 13
    2.3.1 Threat Model and Assumptions 13
    2.3.2 Problem Definition 14

3 Approach 15
  3.1 Overview 15
  3.2 Discovery of Global Function Pointer Calls 16
  3.3 Library Instrumentation 18
  3.4 Symbolic Memory Model 21
    3.4.1 angr’s Default Memory Model 21
    3.4.2 UNTANGLE’s Memory Model 23
  3.5 Modes of Operation 25
    3.5.1 Full Library Mode 25
    3.5.2 Filtered Mode 27
1 | Introduction

Despite its first appearance dating back to the 1970s, binary exploitation is still a relevant problem and a real threat today. As it has already been analyzed in existing academic work [1], the main reason behind the relevance of this problem is the presence of memory corruption vulnerabilities in programs written using memory-unsafe programming languages, like C, which is still indispensable today, and is widely used despite this flaw because of its reliability, portability, and performance.

In the last two decades, exploits for memory corruption vulnerabilities have significantly evolved, as have defenses against them. Most memory corruption exploits are aimed at disrupting the control-flow of a program, which means modifying the execution path of the program so that non-intended instructions are executed. One possible target for an attacker that wants to disrupt the control-flow of a program is the return address of a function saved on its stack [2], which is the address of the first instruction that will be executed upon function return: by overwriting the saved return address, the attacker is able to redirect the control-flow at will. Exploits targeted at such kind of control data can lead to arbitrary code execution: for this reason, they are the target of many new defense proposals.

Recent defense proposals have as their main focus the preservation of the control-flow of a program, so that memory corruption vulnerabilities cannot be exploited to redirect the control-flow on a path that was not originally intended. The main idea behind control-flow preservation is to perform checks to ensure that only allowed execution paths are taken, so that any deviation from them would be recognized as malicious and stopped. An example of state-of-the-art control-flow hijacking defense mechanism is Intel’s Control-Flow Enforcement Technology (CET) [3], which was designed to protect both forward edges (function calls and jumps) and backward edges (function returns) in the Control-Flow Graph (CFG) of a program. Defense mechanisms like CET make it significantly harder for an attacker to gain arbitrary code execution, because they drastically reduce the possible attack surface.

With such defense mechanisms in place, an attacker cannot directly tamper with the
return address of a function: they must target other control variables. **Function pointers** constitute a possible attack surface. A function pointer is a variable that holds the address of a function: this address is not fixed, but can change at runtime. If an attacker manages to overwrite a function pointer with a legitimate address, they can redirect the control-flow of the program without triggering control-flow defenses, possibly gaining arbitrary code execution. This is known as “function pointer hijacking”.

Function pointers can be found in different memory sections of a program or library, and they can even be contained in dynamically allocated objects. We chose to focus on function pointers defined as **global** variables. In particular, we are interested in global function pointers defined in C libraries. Suppose that a library exports a function containing a call to a global function pointer defined within the library, and that such exported function is then used by a program linked against the library: if this program presents a memory corruption vulnerability that allows an attacker to perform arbitrary memory writes (i.e., to write any value to any memory location), it could be used to overwrite the global function pointer and redirect the control-flow of the program once a call to it is reached through the exported library function.

There are however a few complications to this kind of attack. First of all, global function pointers must be found inside a library. Even if the source code of the target library is available, finding all possible global function pointers would require manually inspecting all the source files: this operation alone would require a considerable amount of time. Secondly, the attacker would have to find an interesting call site: a location in the library source code where a global function pointer is called, and that can be reached from one or more exported library functions. This second operation could be performed, similarly to the previous step, through manual inspection of the code. The final step would be to find all the conditions on function parameters and other global variables that, if satisfied, lead the execution path of the target function to the identified call site. Doing all of this by hand, potentially for several different libraries, is a feasible task, but also highly time-consuming and demanding.

With our work, we propose an approach based on static analysis and symbolic execution to automate this whole process given the source code of a C library. Moreover, we identify and solve all the constraints that need to be satisfied in order to reach global function pointer calls at runtime.

We developed **UNTANGLE**¹, a tool that implements this approach with the goal of aiding binary exploitation through global function pointer hijacking. **UNTANGLE** is able to

¹https://github.com/untangle-tool/untangle
perform the operations explained above through three main components:

- An **analyzer** that performs source code analysis on the target library to find global function pointers and their call sites.

- An **instrumenter** that instruments the source code of the target library to prepare it for symbolic execution.

- An **executor** that performs symbolic execution on the instrumented and built library binaries, and employs a custom memory model designed to ease the handling of complex function arguments.

UNTANGLE was tested on several open-source C libraries: `libgnutls`, `libasound`, `libxml2`, `libfuse`, `libcurl`, `libnss`, `libpcre` and `libbsd`. UNTANGLE’s analysis of these libraries found 64 unique global function pointers (57 of which were actually reachable through exported functions) and a total of 1488 exported functions that lead to their calls. Then, it was able to find and verify the correctness of the constraints that need to be satisfied to get to the calls mentioned above for 484 of them.

The main contributions of our work over the state of the art are:

- A methodology to identify global function pointers, their calls sites reachable through exported library functions, and how to reach them.

- UNTANGLE, an automatic tool that implements this methodology.

- An ad-hoc symbolic execution memory model (implemented in UNTANGLE) that helps in dealing with complex objects passed as function parameters.
2 | Background and Motivation

2.1. Background

2.1.1. Exploitation Techniques and Defenses

Memory corruption vulnerability exploits and defenses against them have significantly evolved in the last two decades. A large portion of binary program exploits target the stack memory section of a running program, and involve overwriting the saved return address of the current function call frame. The return address of a function is the memory address of the instruction that will be executed right after the function returns: being able to control its value means being able to control what code will be executed after the function.

One of the earliest memory corruption vulnerabilities is the stack buffer overflow [2, 4], which despite being first documented in the 1970s is still a common vulnerability as of today, listed as CWE-121 by the MITRE Corporation in the Common Weakness Enumeration (CWE) list. A stack buffer overflow (or buffer overrun) allows writing beyond the intended limits of a fixed-size buffer residing in the stack, due to flawed or absent bound checks. If an attacker can write a sufficiently large number of bytes beyond the end of a buffer on the stack, they can overwrite the return address of the current function and gain control over the execution of the program.

One way to execute arbitrary code or commands is through shellcode [4]: a small piece of machine code used as a payload for an exploit, which typically starts a command shell (thus its name), but can also be used to perform different kinds of tasks. A trivial exploit could consist in placing the shellcode in memory and, thanks to a buffer overflow, overwriting the return address of the currently executed function with the address of the shellcode, so that the first instruction executed upon function return will be the first instruction of the shellcode. Several defense mechanisms have been proposed to address this kind of attack. These defense mechanisms move in two main directions: preventing

\[1\]https://cwe.mitre.org/data/definitions/121.html
non-authorized code from being executed or preventing the return address from being overwritten.

Stack canaries were the first attempt at preventing the saved return address from being overwritten. They are used to detect a stack buffer overflow before function return, and thus before the execution of any possibly malicious code. A stack canary is a small value (usually four or eight bytes, respectively on 32-bit and 64-bit systems) randomly generated at the start of each program execution. When a function is called, the stack canary is first copied on the stack right before the saved return address. The function code is then executed, and before returning the canary is read again from the stack and compared with the original value: if the two do not match, a possibly malicious stack buffer overflow happened, and the program aborts execution. This software solution makes it notably harder to exploit a stack buffer overflow vulnerability to gain arbitrary code execution, since a potential attacker would either need to exfiltrate the exact value of the stack canary (for instance through an information leak), or redirect the control-flow of the function by corrupting other important local variables on the stack.

Write Xor Execute (W⊕X) is a memory protection policy according to which any given memory page in a process’s address space can be either writable or executable, but not both. This policy allows to only mark pages that contain actual code as executable (also marking them as non-writable), while keeping pages that should not contain code (e.g., the stack) marked as writable and non-executable. Microsoft implemented its own version of this policy, called Data Execution Prevention (DEP)\(^2\). On modern CPUs, this memory protection policy is also supported as a hardware feature, called No eXecute bit (NX-bit), which allows for fine-grained page permission control at a hardware level. With W⊕X in place, the simple shellcode-based exploit sketched above would not work, because it relies on the attacker being able to inject code into the memory of the process and then execute it: the attacker would only be able to write the shellcode in a writable memory page, but such a page would also be marked as non-executable, preventing the shellcode from being run.

As a response to the proposed defense mechanisms (especially to W⊕X), new exploitation techniques took hold. Return-into-libc [5] makes it possible to avoid code injection, redirecting the control-flow of the program into the C standard library (commonly referred to as “libc”, hence the name of the technique). This technique consists of overwriting the saved return address on the stack with the address of functions that are already present somewhere in the address space of the running program (thus eliminating the need for

code injection), and was proven to be Turing-complete under the right conditions [6]. Libc functions are a good target for redirection, since libc is linked and loaded in memory as a shared library by almost any most program on Unix-based systems: the `system()` function, for instance, can be used to execute arbitrary shell commands. Although the name is “return-into-libc”, this technique also applies to any other shared library loaded by a program at runtime.

2.1.2. Advanced Exploitation Techniques and Defenses

**Return-Oriented Programming** (ROP) [7] is another code-reuse technique that is a more advanced extension of return-into-libc. ROP allows the execution of an arbitrary sequence of instructions in a program without injecting any code. This technique works by means of a “ROP-chain”: a chain of short sequences of instructions, called “gadgets”, that end with a return instruction (thus the name of the technique). ROP gadgets can be found in the code section of the target binary or any shared library loaded by it and thus visible in its address space. By chaining multiple gadgets together, each executing one or more instructions before returning, an attacker can create an arbitrary sequence of machine instructions. Given the right gadgets, ROP is also Turing-complete [8], and can be used to execute arbitrary code. The only limits to this technique are the length of the ROP-chain, limited by the number of bytes that can be written on the stack past the saved return address, and the gadgets available for use, which depend on the specific program and the libraries it uses. ROP obviously defeats W⊕X since all the gadgets involved in the ROP-chain are already present in memory inside pages marked as executable.

Code reuse techniques also include **Jump-Oriented Programming** (JOP) [9] and **Call-Oriented Programming** (COP) [10]. JOP is a code reuse technique that still builds and chains gadgets, which however in this case end with an indirect branch instruction rather than a return instruction. This eliminates reliance on the stack and return instructions (also return-like instructions, e.g., a stack pop followed by a jump to the popped value). COP is a similar code reuse technique that uses gadgets that end in a call instruction.

Once again, new defense mechanisms were developed in order to reduce the impact of these new and more sophisticated techniques. **Address Space Layout Randomization** (ASLR) [11, 12] randomly arranges the address space of a process before starting its execution: the base address of different memory regions (such as the program itself, library code, stack and heap) changes with every new invocation of the same program. ASLR is able to randomize the position of a program in memory only if the program is a **Position-Independent Executable** (PIE), that is, a program that is able to properly
run regardless of its position in memory. For a non-PIE, all the segments specify a fixed memory address where they must always be loaded in order for the program to work. This also means that a non-PIE can be compiled to use absolute memory addresses for memory operations within its sections. Instead, all the memory accesses of a PIE are defined using relative offsets rather than absolute addresses, so that the base address where the program is loaded in memory can be arbitrarily chosen and randomly generated to be different for each execution. This mechanism strongly impacts all the previously discussed exploitation techniques: a ROP-chain cannot be built without knowing the exact address of each gadget. Similarly, a return-into-libc exploit cannot take place without knowing the address of a library function to return to. ASLR is however only effective as long as a potential attacker does not have the ability to somehow leak the address of an interesting memory area (e.g., a section of the program binary itself, a loaded library, etc.). If the exact address of any piece of code and data contained within it can be leaked through vulnerabilities of a program, an attacker would then be able to compute the exact address of any piece of data contained within it (which could be actual data, executable code, or a combination of the two).

With the existence of advanced exploitation techniques, such as the ones that were just described, defense solutions have considerably evolved at a fast pace in recent years. Some solutions are directly targeted at defeating ROP, which is a widely used technique. For instance, ROPdefender [13] is a binary instrumentation tool that enforces return address protection to detect ROP attacks at the cost of performance, adding a runtime overhead equal to the original execution time. ROPecker [14] is another defense mechanism against ROP that detects all the gadgets in a binary program through static analysis, and then detects attacks at runtime by checking the presence of a sufficiently long chain of gadgets in past and future execution flow through execution checkpoints, also supporting detection of non-return gadgets. ROPecker introduces a reasonable runtime performance overhead, and a small memory footprint. Being targeted at ROP, however, neither of these two solutions is capable of detecting and defeating other code-reuse attacks.

Other proposals have preservation of the control-flow of the program as their main objective. Control-Flow Integrity (CFI) [15] is a security policy dictating that software execution must only follow paths of its CFG, which is determined ahead of time through source-code analysis, binary analysis or execution profiling. CFI paved the way for a series of defenses against control-flow hijacking attacks, in the form of both hardware and software solutions.

Intel’s Control-Flow Enforcement Technology (CET) is one of the most recent and advanced CFI enforcement defenses, providing a CPU instruction set architecture exten-
Background and Motivation

sion that allows software to easily set up hardware defenses against ROP, JOP and COP style attacks.

CET has two main features:

- The use of a **Shadow Stack** to provide saved return address protection, preventing ROP.

- **Indirect Branch Tracking** (IBT) to prevent the misuse of indirect branch instructions, typical of JOP/COP attacks.

The shadow stack is a second, hidden stack that is only used by call and return instructions, write-protected using a special combination of memory page permissions. Call instructions push a copy of the return address on the shadow stack (in addition to the “regular” stack). The return instruction pops the return address from both stacks and, if the two do not match, causes an exception preventing a possibly malicious modification of the saved return address on the “regular” program stack.

IBT introduces a new `endbranch` instruction that can be used to mark valid targets for indirect calls and jumps in the program: if an indirect call or jump targets an instruction other than an `endbranch`, the CPU generates an exception and thus prevents any attempt to redirect the control-flow of the program. Therefore, IBT provides forward-edge protection, while the shadow stack provides backward-edge protection. Compilers like **The GNU C Compiler** (GCC) and LLVM’s **clang** are already capable of generating machine code containing `endbranch` instructions at every valid indirect branch/call target.

CET is available on all Intel Core CPUs starting from the 11th generation, and AMD recently announced CET support from its “Series 5000” processors onward. However, CET also needs operating system support. Microsoft already implemented CET support for user-space in Windows, only using the shadow stack, as they already provide indirect-branch protection through “Control-Flow Guard”\(^3\) instead of IBT. As of today, CET is also partially supported by Linux for kernel-space code (IBT support was merged in Linux 5.18), while user-space support is still under development. Because of its accuracy in protecting both forward and backward edges in a CFG, full-CET support in both kernel and user space would make code reuse techniques relying on overwriting the saved return address on the stack (ROP) impossible, and the ones relying on indirect control transfer instructions (JOP, COP) significantly harder, as control-flow would need to be redirected to legitimate targets.

\(^3\)https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard
2.1.3. Function Pointer Hijacking

If an attacker wants to redirect the control-flow of a program, but cannot tamper with the saved return address on the stack because there are protection mechanisms such as CET in place, they must target other kinds of control data, such as function pointers, which are variables (defined either globally or locally) holding the address of a function. As with any other variable, their value can change at runtime, assuming they were defined as non-constant and therefore placed at compile time into a writable section of the program.

Common reasons for function pointer usage in C library code are providing the user with runtime hooks for particular function invocations, implementing function callbacks, and delivering notifications for asynchronous runtime events.

In order to gain arbitrary code execution through the overwrite of a function pointer in a CET-enabled environment, an attacker needs to consider its two main features:

1. If only the shadow stack is active, the attacker can overwrite the function pointer with any address pointing to a memory section containing executable code.

2. If IBT is active (regardless of shadow stack usage), the attacker necessarily needs to overwrite the function pointer with the address of an endbranch instruction. This could be the start of an interesting function, a case of a switch statement compiled using a jump table, or similar. In case the target is a function, the ability to control the parameters supplied to the function could also be necessary (e.g., targeting the system() function provided by the standard C library, one would need to pass the command to run as a parameter), and depends on the specific case at hand.

We focused on the hijacking of global function pointers in C libraries as a possible exploitation entry point, considering that given the right conditions this technique can be used to circumvent even the most modern CFI enforcement techniques like CET.

2.1.4. Static Source Code Analysis

Static analysis is the practice of analyzing a program without executing it, and is an ubiquitous technique for vulnerability research and exploitation. This can be done at the source code level (given the source code of a program or library) or at the binary level (given a compiled program or library), and part of our work focuses on the former. While it can be performed manually, nowadays different platforms, frameworks and tools exist, such as CodeQL⁴, which leverage static source code analysis to detect errors, common

⁴https://codeql.github.com/
2.1.5. Symbolic Execution

Symbolic execution [17] is a dynamic program analysis technique in which the program to be analyzed is driven through its execution by a specialized interpreter, known as symbolic execution engine, with the goal of understanding the inputs needed for the program to produce a specific output or execute a specific part of its code. The symbolic engine feeds the program with symbolic inputs, rather than obtaining concrete inputs from the user or the environment. Symbolic inputs are handled through the use of bitvectors: sequences of boolean variables, each representing a single bit, grouped together to form larger values of the needed bit-length. For example, a 64-bit bitvector can be used to represent a symbolic 64-bit integer.

The engine is then able to track the symbolic inputs given to the program and understand when the program needs to branch based on a condition that depends on the value of some piece of symbolic data. Whenever this happens the engine creates two expressions constraining the symbolic data: one which satisfies the condition and one which does not. Then it duplicates the current state of the program (registers, memory, etc.), and the two initially identical states are advanced in parallel on different sides of the branch: in one state the engine will direct the program to take the branch, while in the other state it will direct the program to not take it. This concept is illustrated in Figure 2.1.

At a high level, a symbolic execution engine can be seen as a “manager” of program
states, keeping track of the symbolic constraints accumulated along the way for each state. These can then be fed to a solver component, typically a Satisfiability Modulo Theories (SMT) [18] solver, which is able to determine if any value satisfying all constraints exists, and if so output one. This operation is known as concretization. At any point during symbolic execution, the engine is able to determine the concrete value for symbolic inputs that would be needed to reach a given state under normal execution. Looking at Figure 2.1, for the state where $\lambda \geq 5$, a valid solution given by the solver would be $\lambda = 10$, from which the engine can determine that a valid input to reach the state would be $x = 10$.

A critical aspect of the design of a symbolic execution engine is its memory model: an engine could implement a memory model that simply lets the program operate on actual memory, it could completely emulate memory operations through symbolic memory, or it could employ a combination of the two. With fully symbolic memory, the engine needs to keep track of the memory contents for each currently handled program state, and is in full control of the data being loaded or stored. When a memory indirection (load/store) operation involving a symbolic address is encountered, the engine has two choices: either perform an actual load/store after concretizing the symbolic address to a concrete value, or delay concretization and turn the result of the load/store into a (possibly highly complex) chain of symbolic If-Then-Else (ITE) expressions to be kept track of and evaluated later.

2.2. State of the Art

Traditional control-flow hijacking attacks target control data such as saved return addresses and function pointers. Data-oriented attacks [19] aim at redirecting the control-flow of a program without tampering with control data, acting only on non-control data, such as variables used by the program to take control decisions. Data-oriented attacks are thus capable of changing the control-flow of a program bypassing defense mechanism that preserve control-flow integrity.

Sophisticated data-oriented attack techniques, along with tools that help automate exploitation, have been proposed in recent years. Data-Oriented Programming (DOP) [20] is a technique to construct expressive non-control data exploits. It allows an attacker to perform arbitrary computations in program memory by chaining the execution of short sequences of instructions, called DOP gadgets. It is a powerful technique, with the downside that the gadget chains must be crafted by hand. Block-Oriented Programming (BOP) [21] is a further improvement of data-oriented attacks: it uses basic blocks as gadgets and leverages symbolic execution to automatically find the constraints
on variables and memory-resident data needed to redirect the control-flow. BOP attacks are specifically aimed at creating a chain of basic blocks that does not trigger CFI preservation mechanisms, and since they do not overwrite the saved return address, they can bypass shadow stacks too. The advantage of BOP, with respect to DOP, is that the gadget chain building process is automated.

To the best of our knowledge, no existing work explores the automation of both global function pointer identification and hijacking in library code. Most of the existing work and research focuses on subsequent exploitation steps instead. In particular, the tool proposed by [21], which automates the creation of BOP-chains, could benefit from our work: one of the requirements for the tool to work correctly is an entry point, i.e., a point from which the tool starts its analysis and constructs the basic block chain. A function pointer that can be overwritten with an arbitrary address would make a good starting point for this kind of analysis.

2.3. Problem Statement

2.3.1. Threat Model and Assumptions

Our exploitation scenario considers a program that is running on a machine employing state-of-the-art control-flow hijacking defenses, such as fully enabled Intel CET. Moreover, the program is also protected through stack canaries, W⊕X memory protection policies and ASLR. We assume that the program uses functions exported by a C library (statically linked or dynamically loaded at runtime) that contain, or can lead to, calls to global function pointers defined within the library itself.

We assume that the program presents a known memory corruption vulnerability that can lead to an arbitrary memory write, also known as “write-what-where” primitive, which gives an attacker the ability to write any value to any writable address. In case of a dynamic library, we also assume that the attacker is able to discover, for example thanks to an information leak, the base address at which the target was loaded under ASLR.

We believe these assumptions are realistic and practical, because they are in line with the ones of the same mechanisms that aim at preventing arbitrary memory reads and writes to be exploited and lead to control-flow hijacking.
2.3.2. Problem Definition

Global function pointers constitute a possible attack surface to gain arbitrary code execution even when advanced control-flow integrity mechanisms like CET are in place. We focus on the problem of finding calls to global function pointers in the source code of a target library and the conditions that would allow us to reach such calls, possibly giving the ability to gain arbitrary code execution.

Commonly used C libraries can be composed of hundreds or even thousands of source code files, while the total number of lines of code can vary from a few thousands to several hundred thousands. Searching for global function pointers and all the locations where they are called by hand is feasible, but not trivial: manually analyzing a large code base would require a considerable amount of time and effort.

Moreover, even if all function pointers and all calls could be found by hand, the hardest part would still be finding the conditions over function parameters and other global variables that would lead the program to the execution of such calls: some libraries contain functions that are hundreds of source code lines long. While manually keeping track of all the conditions that need to be satisfied to reach a certain section of the source code at runtime would be feasible, it would be a demanding, time-consuming and error-prone task. Being able to automate this would therefore be a relevant achievement, as it would make the whole process faster, more practical and more reliable.
3 | Approach

3.1. Overview

The goal of Untangle is to provide precise indications about how to reach calls to global function pointers starting from exported functions of a given C library, including the constraints on function parameters and other global variables that need to be satisfied in order to reach such calls. This is achieved through a combination of static analysis, library source code instrumentation and symbolic execution. The symbolic execution engine we used is angr [22, 23], which employs a fully symbolic memory model, but also permits to extend or override its logic by means of a dynamic plugin system.

The workflow of Untangle is divided into two phases. The **build** phase extracts useful information from the source code of the analyzed library, performs the needed instrumentation, and builds the library to be ready for symbolic execution. The **execution** phase performs the actual symbolic execution, through which the reachability of calls to global function pointers from exported library functions is evaluated.

Untangle involves three main components performing different tasks:

- The **analyzer** creates a CodeQL database for the library and then performs CodeQL queries on it to search for global function pointers, their call sites, and any library function that is able to reach them along with its signature. Additionally, information about struct definitions is also extracted in order to later handle complex structure pointer parameters.

- The **instrumenter** instruments the library source code. It uses the previously extracted information to place a call to an uniquely generated fictitious target function immediately before each identified global function pointer call. These new functions will then be treated as targets for symbolic execution. The modified source code is finally built to generate a binary ready for symbolic execution.

- The **executor** performs symbolic execution on the instrumented and built library using the angr symbolic execution engine. This component also employs two smaller
sub-components: a parser responsible for parsing function signatures and structure definitions into objects to be used during symbolic execution, and a memory sub-component that implements UNTANGLE’s custom memory model (using angr’s plugin system) specifically designed to ease the handling of complex function arguments that are pointers to structure types.

3.2. Discovery of Global Function Pointer Calls

In our work, CodeQL is used to identify global function pointer variables and analyze the call graph of a C library to check whether any calls to such function pointers are potentially reachable from exported library functions.

CodeQL is a powerful framework for static analysis of source code, developed by GitHub, which supports different languages. It operates on ad-hoc databases which can be built from its command line interface starting from a given code base, and it provides the ability to query a database through its own query language similar to SQL. The main purpose for which CodeQL was developed is large scale automation of source code analysis through sets of pre-written queries aimed at discovering potential security vulnerabilities. For example, a query could be written to analyze C source code to check for the presence of buffer overrun vulnerabilities on fixed size buffers.

In order to discover interesting calls done by library code to global function pointers defined within the library, the analyzer creates a CodeQL database for the library.
Then it performs a CodeQL query on the database, which involves three fundamental operations:

1. Discover all existing global variables of type function pointer.

2. Discover call sites for each function pointer that was found. A call site is a C statement in which the global variable is referenced and accessed to make a function call through the function call operator (\(\text{()}\)).

3. Discover all potential entry points to reach each call site. Starting from the enclosing function of a given call site, recursively navigate CodeQL’s call graph\(^1\) backward, from callee to caller, finding all parent functions that are not marked as static, and that can therefore be externally exported by the library.

The first operation is the bare minimum needed to identify whether a library contains any global function pointer or not. The second one ensures that such a global function pointer is used somewhere by library code and not only defined. Finally, the third operation ensures that a call to such a pointer is somehow reachable through a chain of calls starting from a non-static library function. Whether an identified library function is exported or not can then be checked by looking at the exported symbols of the resulting compiled library binaries.

Figure 3.2 shows an example of what was just explained: exported library functions \(F\)

---

\(^1\)https://codeql.github.com/docs/codeql-language-guides/navigating-the-call-graph/
and $G$ would be identified as being able to reach global function pointer call sites (namely $F$ reaches 1 and $G$ reaches 2 and 3) and be considered for further analysis, while $H$ would not.

### 3.3. Library Instrumentation

After global function pointers have been discovered and located in the source code of the library, some preparation steps are needed before starting symbolic execution. Specifically, since UNTANGLE’s end goal is to provide precise information about how to reach any identified global function pointer call, we must be able to provide each individual call as a target to the **executor**. Instrumentation is done at source code level, meaning that modifications are applied to the original library’s source code, which is then compiled and ready to be used.

As already mentioned, CodeQL is used to identify interesting call sites (i.e., specific locations in the source code in terms of file name, line and column). These call sites are extracted based on CodeQL’s own internal **Abstract Syntax Tree** (AST), which is built solely on the source code. CodeQL is therefore not aware of the final compiled library binary and thus does not have any information about the location of actual instructions and basic blocks within it. This is the main problem we face when we want to instrument the code: we cannot target specific instructions (e.g., `call REG`), we can however target exported symbols of the binary.

To achieve this, we would ideally want to naïvely replace any identified function pointer call in the code with the one of a fictitious **TARGET** function.

```c
int (*global_fptr)(int);
int func(int a, int b) {
    int c = global_fptr(a);
    return b + c;
}
```

```c
int TARGET(int _) {return 0;}
int func(int a, int b) {
    int c = TARGET(a);
    return b + c;
}
```

**Listing 3.1**: Example of naïve function pointer call instrumentation

This is however problematic, as we cannot know the functionality of the original function beforehand: the actual function being called can only be known at runtime. Replacing function pointer calls in this way would therefore alter the behavior of library code.
We need to preserve the functionality of the code we are instrumenting to also preserve the reachability of all identified function pointer calls, which is fundamental to provide reliable results. We can accomplish this by creating an auxiliary wrapper macro that expands to a parenthesized expression, evaluating to a call to the target function immediately followed by a call to the original function pointer, and most importantly preserving both the arguments and the return value:

Listing 3.2: Example of bad instrumentation altering original code functionality

Listing 3.3: Example of correct instrumentation preserving original code functionality

We can see an example of this in Listing 3.3. Any optimizing compiler would however eliminate the calls TARGET_foo() and TARGET_bar() as they are no-ops. In order to avoid this kind of compiler optimization, we can put some dummy code with side effects in each target function’s body, like for example an increment of a dummy global exported variable. To also avoid inlining of target functions’ calls, we can use the noinline function attribute.2

Finally, a single function pointer could be called from multiple call sites (e.g., by different

---

functions in the same source file, or even in different files). Since we want to differentiate calls happening at different call sites, we generate a unique ID per call site. The additions made for each function pointer and call site combination are therefore:

- A unique dummy global unsigned variable (NOOP_fptrname_<id>).
- A unique target function (TARGET_fptrname_<id>) marked as noinline and whose body only consists of a single statement incrementing the corresponding global variable, to prevent compiler optimizations from eliminating or inlining calls to it.
- A unique wrapper macro (WRAPPER_fptrname_<id>), used as the actual substitute of the function pointer identifier for this specific call site, evaluating to a call to the target function immediately followed by the original call, preserving the original parameters and return value through the use of an expression.

```c
int (*foo)(int);
int (*bar)(int);
unsigned NOOPT_foo_1;
unsigned NOOPT_bar_1;
void __attribute__((noinline)) TARGET_foo_1(void) {NOOPT_foo_1++;
void __attribute__((noinline)) TARGET_bar_1(void) {NOOPT_bar_1++;
#define WRAPPER_foo_1(...) (TARGET_foo_1(), ({foo(__VA_ARGS__);}))
#define WRAPPER_bar_1(...) (TARGET_bar_1(), ({bar(__VA_ARGS__);}))
int func_one(int x, int y) {
    if (foo && WRAPPER_foo_1(x)) // originally: foo && foo(x)
        if (bar && WRAPPER_bar_1(y)) // originally: bar && bar(x)
            return 0;
    return 1;
}
// Possibly in another file
unsigned NOOPT_foo_2;
void __attribute__((noinline)) TARGET_foo_2(void) {NOOPT_foo_2++;
#define WRAPPER_foo_2(...) (TARGET_foo_2(), ({foo(__VA_ARGS__);}))
int func_two(int x, int y) {
    return x + WRAPPER_foo_2(y); // originally: x + foo(y)
}
```

Listing 3.4: Example of final instrumentation applied to library code

A limitation of this instrumentation approach, further discussed in Chapter 5 “Limitations and Future Work”, is the inability to handle complex macros. CodeQL does not take macro expansion into account, thus if a global function pointer is called from within a macro, CodeQL will report the call as happening at the location where the macro is used. As shown in Listing 3.4, we use a statement expression\(^3\) to encapsulate the origi-

\(^3\)https://gcc.gnu.org/onlinedocs/gcc/Statement-Exprs.html
nal call, so that even macro invocations can be wrapped to instrument function pointer calls happening inside them, but instrumentation will nonetheless fail if the macro is too complex (e.g., consisting of statements that cannot appear in statement expressions).

### 3.4. Symbolic Memory Model

For symbolic execution of a binary program compiled from C source code, operating on pointers to simple primitive C types (e.g., `int *`, `float *`, etc.) means dealing with one level of memory indirection, which is usually simple enough to handle. Pointers to `struct` types on the other hand can be significantly more complex to handle, as structures are often used to hold pointers to other structures, leading to several levels of indirection. How this level of complexity is handled depends on the decisions taken by the symbolic memory model implemented by the symbolic execution engine.

#### 3.4.1. angr’s Default Memory Model

UNTANGLE uses angr as symbolic execution engine. angr’s memory model, already analyzed by existing academic work [24], is fully symbolic: that is, it emulates any memory operation. angr leverages the Z3 SMT Solver [25] to evaluate symbolic addresses for load/store operations. When encountering a symbolic address, angr first evaluates how large the range of values it can assume is. In case of a single possible value, the address is just concretized and the load/store is performed at the concrete address.

In case of multiple possible values though, the behavior differs between load and store operations:

- **For a store operation,** a symbolic address is always concretized to the maximum possible value satisfying its constraints. This is generally useful if the objective of symbolic execution is to find memory corruption bugs in the analyzed program. For example, if an unconstrained 64-bit symbolic pointer is dereferenced for a store of size 8, its value could be concretized to `0xfffffffffffffff8`.

- **For a load operation,** if the range of possible values exceeds a fixed internal threshold, the symbolic address is concretized to an arbitrary value returned by the solver. Otherwise, if the range is small enough, an ITE expression is generated as result of the load, and the address remains symbolic. For example, if a symbolic pointer `p` constrained by a previous check `X ≤ p < Y` is dereferenced to load from memory, and `Y - X` is larger than the concretization threshold, the address can be concretized to any value `v ∈ [X, Y)` at the discretion of the solver.
Figure 3.3: Load/Store handling using angr’s default symbolic memory model

This model presents two main shortcomings:

- Concretizing the symbolic address for a store (e.g., the value of a symbolic pointer passed as function argument) can very easily result in a concrete address that is an invalid memory address (e.g., addressing a memory area that in reality would not be mapped, like 0xfffffffffffffff8).

- Performing a load from a symbolic address can either result in an overly complex ITE expression or in an unpredictable concrete address, chosen at the discretion of the solver, possibly also colliding with addresses of already existing objects.

Both of these problems can impact the chance of successfully completing symbolic execution starting from a library function and symbolically traversing the call chain needed to reach the function pointer calls we are interested in: complex ITE expressions will slow down the solver and increase memory usage, and pointers concretized to invalid addresses can result in failure to reach the wanted target if any sanity check is made by library code along the way.

The most problematic kind of pointers to deal with are pointers to struct types, since as already mentioned structures are often used to hold other pointers, possibly to other structure types, which in turn could hold even more pointers.
### 3.4.2. Untangle’s Memory Model

Passing structures as arguments to library functions through pointers is a very common practice in C, therefore in order to mitigate the issues mentioned in Section 3.4.1 we make use of angr’s plugin system, which among other things also allows for partial or total overriding of the memory model used for symbolic execution.

When it comes to a symbolic pointer used as function parameter, we are ultimately only interested in the contents of the pointed memory, rather than the value of the pointer itself. For pointers to primitive scalar types (\texttt{int *}), double pointers (\texttt{int **}), or pointers to unknown object types (\texttt{void *}) we generally cannot know the pointed object’s size beforehand, as we could be dealing with a single object or an array of objects of arbitrary length (in case of \texttt{void *} we wouldn’t even know the type of the pointed object, let alone its size).

On the other hand, for pointers to \texttt{struct} types, we are most likely dealing with a single object, and this object will also likely contain other pointers to other objects. Untangle’s memory model is designed exactly to manage symbolic pointers to known \texttt{struct} types, leveraging the type information supplied by the \texttt{analyzer} and processed by the \texttt{parser}, in order to significantly reduce the complexity of memory operations during symbolic execution.

When dealing with a symbolic function argument that is a pointer to a known \texttt{struct} type (extracted from the previously created CodeQL database), the \texttt{parser} creates a \texttt{StructPointer} object containing a symbolic bitvector for the actual pointer value and information about the underlying \texttt{struct} fields (offsets and sizes).

```c
struct baz {
    double x, y, z;
};

struct bar {
    int one, two;
};

struct foo {
    int a;
    struct bar *b;
    struct bar *c;
    struct baz d;
    void *e;
};

// Used as function argument
struct foo *foo_ptr;
```

```c
// Used by Untangle’s symbolic memory
// model during symbolic execution
foo_ptr = StructPointer(
    bitvector=BV(...),
    fields={
        0x00: 4, # offset: size
        0x08: StructPointer(
            bitvector=BV(...),
            fields={0x0: 4, 0x4: 4} ),
        0x10: StructPointer(
            bitvector=BV(...),
            fields={0x0: 4, 0x4: 4} ),
        0x18: 24,
        0x38: 8
    }
)
```

Listing 3.5: Example of pointer to struct parsed into a StructPointer object
A nested \texttt{StructPointer} is then created for any field that is also a pointer to a known \texttt{struct} type, and this operation is repeated recursively. The result is a hierarchical data structure (an example of which can be seen in Listing 3.5) containing all the information needed to deal with the symbolic \texttt{struct} pointer, handling any memory indirection through the identified fields.

The \texttt{executor} then uses the bitvector of the root \texttt{StructPointer} object as symbolic function argument, and keeps track of all the \texttt{StructPointer} objects created, so that the usage of any of their bitvectors during any load/store memory operation can be recognized and handled properly. The first load/store operations that uses the symbolic bitvector of a tracked \texttt{StructPointer} \( p \) will concretize its value to a known address. A chunk of memory of the needed size is then reserved at this address to hold the contents of the underlying \texttt{struct} that \( p \) is tracking, and the bitvector for any nested \texttt{StructPointer} field of \( p \) is stored at the correct offset in the memory chunk. The load/store operation to the now-concrete address is then forwarded to \texttt{angr}'s default handler, like any other load/store operation.

This early concretization of symbolic pointers allows for both faster and more accurate symbolic execution, as well as coherent final evaluation of \texttt{struct} pointer function parameters. After symbolic execution terminates, the \texttt{executor} recursively evaluates each field of all the involved \texttt{StructPointer} objects, and provides a human-readable and easy to understand output, an example of which is shown in Listing 3.6.
3.5. Modes of Operation

UNTANGLE leverages symbolic execution to find the constraints on the parameters of exported functions and on global variables that need to be satisfied to reach a call to a function pointer. UNTANGLE provides two symbolic execution modes: full library execution and filtered execution.

3.5.1. Full Library Mode

In full library mode the executor symbolically executes all the identified library functions that can potentially lead to global function pointer call sites one by one. Each run stops at the first call site that can be reached, regardless of the location of the call site and the name of the called function pointer.

Before symbolic execution starts, the executor produces a list of all the exported functions that contain, or can lead to, a call to a global function pointer. Then, in order to setup symbolic execution, it performs the following steps for each exported function:

- Parse the function signature to extract the number of parameters and their types.
- Create a symbolic bitvector for each parameter that was identified in the previous step.
- Mark the entire .data and .bss sections as symbolic. Since one of our goals is
to also identify constraints on global variables, which live in either `.data` or `.bss` depending on whether they are initialized or not. This step is needed to later verify whether any portion of these memory regions is involved in any constraints, which wouldn’t happen by default, as `angr` would consider `.data` and `.bss` concrete and load their contents from the analyzed binary.

- Produce a list of target addresses to be reached through symbolic execution: these are the memory addresses of the `TARGET_...` symbols generated and inserted during library instrumentation (as shown in Listing 3.4).

Finally, the `executor` symbolically executes the library function using `angr`. Since we provide all known call sites as targets, symbolic execution will stop as soon as it reaches any of them, right before a call to a global library function pointer is about to happen.

The output of `UNTANGLE` is, for each exported function, a file containing:

- The name of the function pointer whose call was reached through symbolic execution (if any), together with its location in the code.
- The values that the function arguments (if any) should have to reach the call.
- The value that any identified global variable (involved in the constraints to reach the call) should have, together with its offset in the library binary, the section in which it is found, and its size.
- A list of all the constraints found by `angr`.
- How much time and memory were used to perform symbolic execution of the function.

`UNTANGLE` allows specifying a timeout and a memory limit for the execution of each function through command line options. If any of these two limits is exceeded, if no solution can be found, or if an error happens that prevents symbolic execution from terminating, the output file contains a message explaining what went wrong.

Finally, `UNTANGLE` also provides a command line option to automatically verify the obtained result (in case symbolic execution finds a solution) by compiling and running a small test program, as we explained in more detail in Section 4.3.1 “Evaluation of Symbolic Execution Results”.
3.5.2. Filtered Mode

In filtered mode, Untangle lets the user specify zero or more filters (as regular expressions) for the name of the library function to execute, the name of the global function pointer, and the specific call site location to reach. The executor will then only target the matched library functions, function pointers and call sites. This mode allows for fine-grained control of the symbolic execution, giving the ability to select any possible path from identified exported library functions to identified global function pointer call sites, and is useful to explore more options in case full library mode reaches a call site that for some reason is unacceptable in a specific scenario.

Figure 3.5 shows a comparison of full library versus filtered execution mode: in both instances the same exported library function is being symbolically executed, but in the second instance only a single call site was selected.

![Figure 3.5: A single exported function being symbolically executed in full library execution mode versus filtered execution mode](image-url)
4 | Experimental Validation

4.1. Goals

In order to test UNTANGLE we performed full library execution tests on multiple C libraries commonly used on GNU/Linux systems. The main focus of testing was the symbolic execution phase: success rate of symbolic execution (i.e., what percentage of runs is able to find and return a solution), validity of found solutions, and amount of system resources needed to find them. We collected statistics about the quantity and validity of symbolic execution results, then about performance in terms of execution time and memory usage.

Due to its nature, the static analysis phase cannot be programmatically tested: whether all “interesting” function pointers can actually be identified or not can only be proven by manual inspection of library source code. The few instances where global function pointers were identified as present, but not as reachable through a chain of calls starting from an exported library function, were manually checked to rule out false negatives (taking into account the limitations explained in Chapter 5). The actual identification of the presence of global function pointers, done through a simple CodeQL query, depends on the reliability of CodeQL itself and was out of the scope of our tests.

The correctness of symbolic execution runs that result in not finding any solution depends on the reliability of the engine being used: angr in our case. Proving that call sites reported as unreachable by angr were truly unreachable due to the constraints encountered along the path was also out of the scope of our tests, as it requires prolonged manual analysis of library source code for each instance, and the total number of instances is prohibitive. However, the validity of found solutions, as already mentioned, is testable and was in fact tested.
4. Experimental Validation

<table>
<thead>
<tr>
<th>Library</th>
<th>Estimated lines of source code</th>
<th>Unique global function pointers</th>
<th>Reachable function pointers</th>
<th>Unique call sites</th>
<th>Exported functions able to reach any call site</th>
<th>Unique paths to call sites</th>
</tr>
</thead>
<tbody>
<tr>
<td>libgnutls v3.6.16</td>
<td>422,804</td>
<td>15</td>
<td>14</td>
<td>1,338</td>
<td>827</td>
<td>29,817</td>
</tr>
<tr>
<td>libasound v1.2.4</td>
<td>94,288</td>
<td>3</td>
<td>2</td>
<td>383</td>
<td>243</td>
<td>7,739</td>
</tr>
<tr>
<td>libxml2 v2.9.10</td>
<td>353,481</td>
<td>8</td>
<td>6</td>
<td>2,125</td>
<td>225</td>
<td>254,096</td>
</tr>
<tr>
<td>libfuse v3.11</td>
<td>21,568</td>
<td>1</td>
<td>1</td>
<td>110</td>
<td>110</td>
<td>110</td>
</tr>
<tr>
<td>libcurl v7.84</td>
<td>152,921</td>
<td>5</td>
<td>5</td>
<td>271</td>
<td>48</td>
<td>11,238</td>
</tr>
<tr>
<td>libnss v2.31</td>
<td>10,568</td>
<td>21</td>
<td>18</td>
<td>34</td>
<td>15</td>
<td>74</td>
</tr>
<tr>
<td>libpcre v8.39</td>
<td>107,530</td>
<td>3</td>
<td>3</td>
<td>13</td>
<td>12</td>
<td>36</td>
</tr>
<tr>
<td>libbsd v0.11.3</td>
<td>11,316</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1,174,476</strong></td>
<td><strong>64</strong></td>
<td><strong>57</strong></td>
<td><strong>4,282</strong></td>
<td><strong>1,488</strong></td>
<td><strong>303,118</strong></td>
</tr>
</tbody>
</table>

Table 4.1: List of tested libraries and relevant statistics

4.2. Dataset

The tested C libraries were chosen among the top-ranked free open-source libraries listed under the “libs” section of the Debian package Popularity Contest\(^1\), using the latest version provided by Debian 11 packages. A first check was made through CodeQL with a simpler query to identify whether global function pointers were present at all, and if so Untangle was run to analyze the library. In total, around 50 libraries were manually downloaded, compiled, and checked: 8 of those, listed in Table 4.1, turned out to be of interest and were tested.

4.3. Experimental Setup

As can be seen in Table 4.1, the number of unique code paths starting from exported library functions and possibly leading to a global function pointer call can be quite large. For this reason, we did not test every single path, as the amount of time needed for such kind of analysis would have been prohibitive, but rather focused on analyzing the reachability of any global function pointer call starting from every single exported function.

4.3.1. Evaluation of Symbolic Execution Results

An important step after finding a solution through symbolic execution is to check its validity. This can be done by compiling and running a test C program that uses the solution found through symbolic execution to appropriately set up a function call to the

\(^1\)https://popcon.debian.org/main/index.html
tested library function. This is however not a simple task, and depending on the library, the test program would need to be significantly complex in order to correctly compile. Using a library function means importing the right header files, creating variables of the appropriate type and value (which can in turn require additional headers for the type definitions), and also linking the right library binary after compilation. Doing this requires multiple steps that change based on the specific library, and cannot be easily done programmatically. We have therefore implemented a simpler automatic verification method that involves the use of libdl\(^2\) to dynamically load instrumented libraries at runtime and The GNU Debugger (GDB) to monitor whether identified call sites are reached through automatically inserted breakpoints.

This naïve automatic validation mechanism is a built-in feature of Untangle. The goal is to avoid false positive results: if execution reaches a breakpoint set at the target call site while running under GDB, the found solution must inevitably be correct (it could be trivial, but nonetheless correct). It can however yield false negatives: functions for which we found a solution, but through which the global function pointer call site identified during symbolic execution is not reached during automatic validation. These are more complex to handle and require manual testing in order to be identified.

Automatic validation consists of the following steps performed after a successful symbolic execution run (i.e., one that found a satisfiable solution):

1. Compile a simple C program through a pre-defined template providing information about the library to dynamically load, the exported function to call, and the needed values for its parameters. This program will:

   (a) Load the instrumented library through libdl.

   (b) Copy the value of any needed structure evaluated during symbolic execution into a memory region mapped exactly for this purpose.

   (c) Call the exported library function to test with the right parameters, where any pointers to structures will be pointing into the previously mapped memory region.

2. Run the compiled program under GDB, setting a breakpoint on the TARGET function (added at instrumentation time) corresponding to the global function pointer call site that needs to be reached.

3. Check whether the breakpoint is reached or not.

\(^2\)https://refsspecs.linuxbase.org/LSB_3.0.0/LSB-generic/LSB-generic/libdlman.html
4.3.2. Performance Evaluation

The execution time and the memory usage for symbolic execution, as well as the overall time spent analyzing a given library, are important metrics to measure Untangle’s performance. The machine used for testing is equipped with a 64-bit Intel Core i9-10900 CPU (base core clock speed of 2.80GHz), 32GiB of RAM, and runs Debian 11 GNU/Linux v5.10. Libraries were therefore compiled for Linux x86-64 using GCC version 10.2.1, the standard compiler for Debian GNU/Linux systems. Where possible and permitted by library configuration scripts, the optimization option chosen was -O2, and the use of advanced CPU-specific instruction sets (such as AVX2, SSE4, etc.) was disabled to avoid issues with PyVEX\(^4\) [26], the Python library used by angr for translation of machine instructions.

Since angr does not offer multi-threading support, all performed symbolic execution runs consist of single-threaded processes. Each symbolic execution run was limited to 15 minutes of time and 16GiB of RAM usage (Resident Set Size). Runs exceeding any of the two limits were halted, while still collecting resource usage information for statistical purposes.

4.4. Results

As can be seen in Table 4.2, a solution was found through symbolic execution for 58.9% (877) of the 1488 total exported library functions analyzed. The overall average execution time for a single run, including those that timed out, was 1 minute and 36 seconds. The overall average memory usage for a single run, including those that ran out of memory, was around 4.27GiB.

Moving on to validation, for which the results are summarized in Table 4.3, out of the 877 solutions found, 484 of those (55.2% of the found solutions, 32.5% of the total tests) were proven to be valid using the method described in Section 4.3.1. We can also notice the result of what we explained in Chapter 3, Section 3.4.1 “angr’s Default Memory Model”: instances where pointers to primitive types need to be passed as function arguments can be concretized by angr to invalid memory addresses, which can make automatic validation fail. Due to this reason, even if Untangle was able to find a solution that did not pass validation, there’s a chance that such an instance is a false negative. Untangle will report the solution anyway, but manual testing is needed to understand additional and possibly more complex constraints that weren’t automatically identified.

\(^3\)We use Mebibytes and Gibibytes as memory size units: 1GiB = 1024MiB
\(^4\)https://github.com/angr/pyvex
Finally, looking at runs that did not result in a found solution, we can break down the reason into four categories, which are shown in Table 4.4:

1. **Unreachable**: symbolic execution completed, but the engine determined that none of the identified call sites is reachable. Apart from limitations of the angr engine and solver, this can simply happen because the constraints leading to call sites are impossible to satisfy.

2. **Timeout**: the run was halted after exceeding the set timeout of 15 minutes.

3. **Memory**: the run was halted after exceeding the set memory usage limit of 16GiB.

4. **Engine error**: the run was halted because of an internal error of the symbolic execution engine. This can happen for multiple reasons, the most common of which are constraints that become too complex (for example causing the solver to exceed Python’s maximum call stack size) or bugs in the engine code.

### Table 4.2: Symbolic execution results and resource usage

<table>
<thead>
<tr>
<th>Library</th>
<th>Tested functions</th>
<th>Symbolic execution solution</th>
<th>Runtime</th>
<th>Average memory usage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Found</td>
<td>Not found</td>
<td>Total</td>
</tr>
<tr>
<td><strong>libgnutls</strong></td>
<td>827</td>
<td>460 (55.6%)</td>
<td>367 (44.4%)</td>
<td>20h 21m</td>
</tr>
<tr>
<td><strong>libasound</strong></td>
<td>243</td>
<td>153 (63.0%)</td>
<td>90 (37.0%)</td>
<td>7h 52m</td>
</tr>
<tr>
<td><strong>libxml2</strong></td>
<td>225</td>
<td>139 (61.8%)</td>
<td>86 (38.2%)</td>
<td>7h 29m</td>
</tr>
<tr>
<td><strong>libfuse</strong></td>
<td>110</td>
<td>59 (53.6%)</td>
<td>51 (46.4%)</td>
<td>2h 53m</td>
</tr>
<tr>
<td><strong>libcurl</strong></td>
<td>48</td>
<td>40 (83.3%)</td>
<td>8 (16.7%)</td>
<td>52m 50s</td>
</tr>
<tr>
<td><strong>libnss</strong></td>
<td>15</td>
<td>9 (60.0%)</td>
<td>6 (40.0%)</td>
<td>31m 15s</td>
</tr>
<tr>
<td><strong>libpcre</strong></td>
<td>12</td>
<td>9 (75.0%)</td>
<td>3 (25.0%)</td>
<td>36s</td>
</tr>
<tr>
<td><strong>libbsd</strong></td>
<td>8</td>
<td>8 (100%)</td>
<td>0</td>
<td>8s</td>
</tr>
<tr>
<td><strong>Overall</strong></td>
<td>1 488</td>
<td>877 (58.9%)</td>
<td>611 (41.1%)</td>
<td>40h</td>
</tr>
</tbody>
</table>

### Table 4.3: Validation of successful symbolic execution runs

<table>
<thead>
<tr>
<th>Library</th>
<th>Tested functions</th>
<th>Solution found</th>
<th>Validation result</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Pass</td>
<td>Fail</td>
</tr>
<tr>
<td><strong>libgnutls</strong></td>
<td>827</td>
<td>460 (55.6%)</td>
<td>272 (32.9%)</td>
<td>188 (22.7%)</td>
</tr>
<tr>
<td><strong>libasound</strong></td>
<td>243</td>
<td>153 (63.0%)</td>
<td>91 (37.4%)</td>
<td>62 (25.5%)</td>
</tr>
<tr>
<td><strong>libxml2</strong></td>
<td>225</td>
<td>139 (61.8%)</td>
<td>60 (26.7%)</td>
<td>79 (35.1%)</td>
</tr>
<tr>
<td><strong>libfuse</strong></td>
<td>110</td>
<td>59 (53.6%)</td>
<td>15 (13.6%)</td>
<td>44 (40.0%)</td>
</tr>
<tr>
<td><strong>libcurl</strong></td>
<td>48</td>
<td>40 (83.3%)</td>
<td>30 (62.5%)</td>
<td>10 (20.8%)</td>
</tr>
<tr>
<td><strong>libnss</strong></td>
<td>15</td>
<td>9 (60.0%)</td>
<td>2 (13.3%)</td>
<td>7 (46.7%)</td>
</tr>
<tr>
<td><strong>libpcre</strong></td>
<td>12</td>
<td>9 (75.0%)</td>
<td>6 (50.0%)</td>
<td>3 (25.0%)</td>
</tr>
<tr>
<td><strong>libbsd</strong></td>
<td>8</td>
<td>8 (100%)</td>
<td>8 (100%)</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>1 488</td>
<td>877 (58.9%)</td>
<td>484 (32.5%)</td>
<td>393 (26.4%)</td>
</tr>
</tbody>
</table>

Finally, looking at runs that did not result in a found solution, we can break down the reason into four categories, which are shown in Table 4.4:
As we can notice by looking at Table 4.4, the first category is actually the least common. The most common failure reason is running out of memory. 16GiB is a reasonable amount of RAM; exceeding it is indicative of accumulating too many symbolic states along the way, which ultimately also results in slower running times.
5 | Limitations and Future Work

The main limitations of Untangle come from limitations of the tool used for static analysis of source code: CodeQL. As mentioned in Section 3.2, CodeQL performs its analysis at the source code level, and therefore does not provide any information about the location of specific instructions or basic blocks in the resulting compiled binaries.

First of all, while it surely speeds up the search for global function pointer call sites in library source code with respect to manual inspection, Untangle is not always able to identify all of the possible call sites. Instances where a call happens indirectly (and not through the global function pointer identifier) are not detected: for example, global function pointers that are first copied into local variables, which are then used to perform the actual call later in the code, perhaps in a different function. Detecting and correctly handling such cases would require tracking assignments and copies of variables throughout the entire code base. CodeQL offers a mechanism to do this through taint analysis, but would not still be able to cover all instances. We found an example of this limitation in glibc\(^1\) (GNU implementation of the C standard library) for the `__malloc_hook` global function pointer, used to hold the address of a function that can be executed as a replacement of the `malloc()` library function. As shown in Listing 5.1, the address contained in `__malloc_hook` is loaded into another local variable using inline assembly, and the latter is then used to perform the actual call. This kind of indirect variable assignment is enough to make the call undetectable, as CodeQL does not handle inline assembly.

Another possible limitation of Untangle is the way instrumentation is performed. Depending on how the library is written, not every call found by the analyzer might be replaceable with another function call or macro invocation. The reason behind this is that if a function pointer is referenced within a macro definition, CodeQL will detect the line in the code where the macro is invoked as call site. Not every macro that involves a global function pointer is a simple redefinition of the function pointer, though: a macro could expand into a series of statements, a complex expression, or even partial pieces of

\(^{1}\)https://www.gnu.org/software/libc
Listing 5.1: __malloc_hook function pointer call in glibc’s malloc()

code that only make sense in a specific context. We found an example of this in gnutls\textsuperscript{2}, a secure communication library that implements SSL, TLS and DTLS protocols.

The code in Listing 5.2 shows an example of a macro that is too complex to be wrapped into a statement-expression as described in Section 3.3, because it is part of a switch/case construct. CodeQL finds the reference to CASE_SEC_PARAM because it invokes _gnutls_cert_log, which is another macro containing the actual call to the global function pointer gnutls_free. The invocation of a macro of this kind cannot be correctly instrumented by Untangle: the result would be invalid C code. Therefore, in order to analyze this library, one would have to manually expand every instance of the macro before instrumenting it (which is what we did to test this library).

The information provided by CodeQL makes the location of global function pointer call sites only identifiable at the source code level. Being able to extract call site locations in the compiled library binaries would remove the need to perform instrumentation of the source code, and allow for it to be performed at a later compilation stage. Frameworks like the LLVM Compiler Infrastructure\textsuperscript{3} that provide introspection and instrumentation ability at the Intermediate Representation (IR) level or even at the machine code level could be leveraged to directly instrument the generated code. Additionally, being able

\textsuperscript{2}https://gitlab.com/gnutls/gnutls
\textsuperscript{3}https://llvm.org/
to keep track of the offset within the .text section of the generated call instruction for each interesting global function pointer call site, one could simply provide those directly to angr as target for symbolic execution.

Finally, because of its design, UNTANGLE needs the source code of the library to analyze. An improvement possibility that could be explored is the extension of our approach to binaries with no source code available. Frameworks such as Joern [27], that enable static analysis of binary executables, could be leveraged along with heuristics to identify which call sites to consider as global function pointer calls.
6 Conclusions

This work aimed at providing an automated methodology for finding global function pointers whose calls are reachable through exported functions of a C library, along with all the constraints that need to be satisfied to reach them.

The approach we presented employs static analysis of the source code of a target library to identify global function pointer calls and interesting exported functions, combined with symbolic execution of the latter to find constraints on function parameters and global variables that need to be satisfied in order to reach such calls.

We developed UNTANGLE\(^1\), a tool that implements this approach with the aim of assisting manual binary exploitation through function pointer hijacking. UNTANGLE relies on an ad-hoc symbolic execution memory model that makes it possible to deal with complex objects, such as pointers to structures, passed as function parameters.

The results we obtained from the tests run on UNTANGLE show that global function pointers can be found in commonly used C libraries and that, under the right conditions, it is possible to reach calls to them starting from exported library functions. Even with the most restrictive state-of-the-art control-flow integrity measures in place, such variables offer a possibility to gain arbitrary code execution if they are overwritten with the address of a carefully chosen legitimate target. Therefore, UNTANGLE provides a reasonable and practical exploitation aid for function pointer hijacking.

\(^1\)https://github.com/untangle-tool/untangle
Bibliography


## List of Figures

2.1 Symbolic execution compared to normal execution .................................. 11

3.1 Architecture overview of Untangle ....................................................... 16

3.2 Discovery of “interesting” exported library functions .............................. 17

3.3 Load/Store handling using angr’s default symbolic memory model ........... 22

3.4 Load/Store handling using Untangle’s memory model ............................ 24

3.5 A single exported function being symbolically executed in full library exec-
    ution mode versus filtered execution mode ........................................ 27
Listings

3.1 Example of naïve function pointer call instrumentation . . . . . . . . . . 18
3.2 Example of bad instrumentation altering original code functionality . . . 19
3.3 Example of correct instrumentation preserving original code functionality . 19
3.4 Example of final instrumentation applied to library code . . . . . . . . . . . . 20
3.5 Example of pointer to struct parsed into a StructPointer object . . . . . . 23
3.6 Example of UNTANGLE output showing evaluated StructPointer contents 25
5.1 __malloc_hook function pointer call in glibc’s malloc() . . . . . . . . . . 36
5.2 Example of complex macro that UNTANGLE is unable to instrument. . . 37
# List of Tables

4.1 List of tested libraries and relevant statistics ........................................ 30
4.2 Symbolic execution results and resource usage ....................................... 33
4.3 Validation of successful symbolic execution runs ..................................... 33
4.4 Break-down of unsuccessful symbolic execution runs ................................. 34