Untangle: Aiding Global Function Pointer Hijacking for Post-CET Binary Exploitation

Alessandro Bertani*, Marco Bonelli, Lorenzo Binosi, Michele Carminati, Stefano Zanero, and Mario Polino

Politecnico di Milano
{alessandro.bertani, lorenzo.binosi, michele.carminati, stefano.zanero, mario.polino}@polimi.it
marco.bonelli@mail.polimi.it

Abstract. In this paper, we combine static code analysis and symbolic execution to bypass Intel's Control-Flow Enforcement Technology (CET) by exploiting function pointer hijacking. We present Untangle, an open-source tool that implements and automates the discovery of global function pointers in exported library functions and their call sites. Then, it determines the constraints that need to be satisfied to reach those pointers. Our approach manages naive built-in types and complex parameters like structure pointers. We demonstrate the effectiveness of Untangle on 8 of the most used open source C libraries, identifying 57 unique global function pointers, reachable through 1488 different exported functions. Untangle can find and verify the correctness of the constraints for 484 global function pointer calls, which can be used as attack vectors for control-flow hijacking. Finally, we discuss current and future defense mechanisms against control-flow hijacking using global function pointers.

Keywords: Binary Exploitation · Control-Flow Integrity · Control-Flow Hijacking · Static Analysis · Symbolic Execution

1 Introduction

Binary exploitation is a significant problem and threat due to memory corruption vulnerabilities [36] in programs written using memory-unsafe languages like C. Despite this flaw, C is still widely used for its reliability, portability, and performance. Most memory corruption exploits aim to disrupt a program's control flow. Recent defense proposals primarily focus on preserving the control flow, to prevent memory corruption vulnerabilities from being exploited to redirect it on an unintended path. 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 a state-of-the-art control-flow hijacking defense mechanism is Intel's **Control-Flow Enforcement Technology** (CET) [33], which was designed to protect

* Corresponding author

both forward edges (function calls and jumps) and back 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, as 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 but must target other control variables like **function pointers**, which can be found in different memory sections of a program or library and constitute a possible attack surface.

This work focuses on **global** function pointers defined in C libraries. Global function pointers are easy to identify and find in process memory, and finding such an attack vector in a widespread library makes the approach generic, enabling the exploitation of binaries having the library as a dependency. Finding global function pointers in C libraries would simplify exploit writing in CET-enabled scenarios and would be helpful to C library developers to detect the presence of such attack vectors. To better understand how an attacker can exploit function pointers, consider a library that exports a function containing a call to a global function pointer defined within the library. This exported function is then used by a program using the library. If this program presents an arbitrary write vulnerability (i.e., a vulnerability that allows the attacker to write any value to any memory location), it can 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 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, one would need to manually analyze it to find all possible global function pointers, interesting call sites, and all the conditions leading the execution to them. Doing all this by hand, potentially for several different libraries, is a feasible but highly time-consuming and demanding task. Our work proposes an approach based on static analysis and symbolic execution [25] 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 to reach global function pointer calls at runtime.

We present UNTANGLE[1], an open-source tool that implements the proposed approach to aid binary exploitation through global function pointer hijacking. It is important to highlight that UNTANGLE is also helpful from a defense perspective since it helps C library developers to discover the identified attack surface and library users to detect affected libraries. UNTANGLE performs its task through four main components: the *Global Pointers Extractor*, the *Instrumenter*, the *Parser* and the *Executor*. The *Global Pointers Extractor* performs source code analysis on the target library to find global function pointers and their call sites. The *Instrumenter* instruments the source code of the target library to prepare it for symbolic execution. The *Parser* extracts information on structure types definitions and function signatures to improve the symbolic execution process. The *Executor* performs symbolic execution on the instrumented library binaries and employs a custom memory model designed to ease handling

[1] https://github.com/untangle-tool/untangle

complex function arguments. We evaluate UNTANGLE on several open-source C libraries (i.e., `libgnutls`, `libasound`, `libxml2`, `libfuse`, `libcurl`, `libnss`, `libpcre` and `libbsd`). UNTANGLE identifies 64 unique global function pointers (57 of which are reachable through exported functions) and 1488 exported functions that lead to their calls, finding and verifying the constraints' correctness to satisfy those calls in 484 cases. In summary, the contributions are the following:

- 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 end-to-end. It takes the source code of a library as input and produces as output all the function pointers found inside the library, which are reachable through exported functions and concrete parameter values that satisfy the conditions that allow it to reach them.
- An ad-hoc symbolic execution memory model (implemented in UNTANGLE) that deals with `struct` pointers passed as function parameters.

2 Background

Static Code Analysis. Static code analysis is the practice of analyzing a program without executing it, and is a widely adopted technique for vulnerability research. It can be performed at the source code level (given the source code of a program or a library) or at the binary level (given the compiled program or library). In our work, we perform static source code analysis to identify global function pointers in library code. To perform this task, we use CodeQL [2], a static analysis framework developed by GitHub, that provides a formal query language to specify the targets of the static analysis process.

Symbolic Execution with `angr`. Symbolic execution 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*. The engine feeds the program with *symbolic inputs*, rather than concrete inputs obtained by the user or the environment. Whenever the analyzed program needs to evaluate a branch condition involving symbolic data, the engine creates two expressions constraining the symbolic data: one that satisfies the condition and one that does not. Then, it duplicates the current state of the program, and two initially identical states are advanced in parallel on the two different sides of the branch, keeping track of the constraints on symbolic variables that caused the state duplication. A critical aspect of a symbolic execution engine is its *symbolic memory model*, which defines the policies for managing memory accesses. Because of its Python-based interface, flexibility, and modular plugin system, we chose `angr` [1, 35, 37] as a symbolic execution engine. `angr`'s memory model, already analyzed in previous works [12], is fully symbolic, i.e., it emulates every memory operation by concretizing memory addresses whenever it is needed.

When dealing with a symbolic address, at first `angr` evaluates how large the range of values it can assume is. In the case of a single possible value (depending on the constraints present in the current state), the address is concretized and the load/store is performed at the concrete address. However, in the case of multiple possible values, 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 can be useful if the objective of symbolic execution is to find memory corruption bugs in the analyzed program. For instance, 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 If-Then-Else expression is generated and the address remains symbolic. The issue with the first case is that unpredictable concrete addresses could be generated, which will likely be colliding with the addresses of other existing objects. These issues can impact the chance of successfully traversing the call chain needed to reach the calls to function pointers we are interested in during symbolic execution. Complex data types, such as pointers to structure, are an especially problematic case: `angr` has no knowledge about struct sizes, and this can cause instances where addresses of different struct pointers are concretized to contiguous values. This is likely to cause memory overlaps and generate invalid results.

2.1 Exploitation Techniques and Defenses

Return-Oriented Programming (ROP) [13, 16, 28, 29]. It is a code-reuse technique that allows the execution of an arbitrary sequence of instructions in a program without injecting any code. This technique uses 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 [21] and can execute arbitrary code. The only limits to this technique are the length of the initial 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 defeats defense mechanisms such as **Write Xor Execute** ($W \oplus X$) since all the gadgets involved in the ROP-chain are located in executable memory pages. Code reuse techniques also include **Jump-Oriented Programming** (JOP) [11] and **Call-Oriented Programming** (COP) [30]. JOP is a code reuse technique that builds and chains gadgets that end with an *indirect branch* instruction rather than a return instruction. This eliminates reliance on the stack and 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.

Defense mechanisms directly affect the impact of code-reuse techniques. **Address Space Layout Randomization** (ASLR) [9, 32] 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 execution of the same program. ASLR can randomize the position of a program in memory only if the program is a **Position-Independent Executable** (PIE), that is, a program that can properly run regardless of its position in memory. 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 the previously discussed exploitation techniques: a ROP-chain cannot be built without knowing the exact address of each gadget. ASLR is, however, only effective as long as a potential attacker cannot *leak* the address of an interesting memory area (e.g., a section of the program binary itself, a loaded library). 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, as offsets inside the binary are fixed. Recent defense solutions are directly targeted at defeating ROP: some examples are kBouncer [27], ROPdefender [20] and ROPecker [18]. While targeted towards ROP, however, neither of these solutions can detect and defeat other code-reuse attacks. It has been shown in previous works that these defenses have some shortcomings and can be bypassed with low effort [15, 31]. Other proposals target the preservation of the control flow of a program rather than the mitigation of a specific exploitation technique. **Control-Flow Integrity** (CFI) [8, 10, 14, 26] 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 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 extension that allows the software to easily set up hardware defenses against ROP, JOP and COP style attacks. CET has two main features: ① the use of a **Shadow Stack** [19] to provide saved return address protection, preventing ROP; ② **Indirect Branch Tracking** (IBT) [27] to prevent the misuse of indirect branch instructions, typical of JOP/COP attacks. 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, operating systems' support towards CET is still partial. Because of its accuracy in protecting both forward and back 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) harder, as control-flow would need to be redirected to legitimate targets, identified by *endbranch* instructions.

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**. 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. To gain arbitrary code execution through a function pointer in a CET-enabled environment, an attacker needs to consider two main possible scenarios. In the first scenario, when only the shadow stack is active, the attacker can overwrite the function pointer with any address pointing to a memory section containing executable code. In the second scenario, when 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 focus on hijacking 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 Intel CET.

3 Related Work

Non-control data attacks [17, 22] are state-of-the-art binary exploitation techniques, and are a viable alternative to "traditional" control-flow hijacking attacks. They aim at redirecting the program's control flow without tampering with control data, acting only on non-control data, such as variables used by the program to make control decisions. Data-oriented attacks are thus capable of changing the control flow of a program by bypassing defense mechanisms that preserve control-flow integrity. Sophisticated non-control data attack techniques and tools that help automate exploitation have been proposed in recent years. **Data-Oriented Programming** (DOP) [23] 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) [24] 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. These techniques, however, have their limitations: they are complex and only work in particular situations. Global function pointer hijacking requires less effort and is a viable alternative to perform binary exploitation in specific settings.

Discussion. 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 BOP Compiler (BOPC) [24], 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 be a good starting point for this analysis.

4 Threat Model and Problem Statement

Our exploitation scenario considers a program 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, $W \oplus X$ 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 the case of a dynamic library, we also assume that the attacker can discover, for example, thanks to an information leak, the base address at which the target was loaded under ASLR. These assumptions are realistic and practical since they are in line with the ones of the mechanisms that aim at preventing arbitrary memory reads and writes from being exploited.

Motivation and Research Goal. One of the fundamental steps while writing an exploit that aims at gaining arbitrary code execution is to gain control of the instruction pointer. This is usually achieved by overwriting the saved return address of a function on the stack, by overwriting a function pointer contained in an object on the stack or on the heap (e.g., a *vtable* pointer), or by overwriting global function pointers (e.g., in shared libraries). If CFI enforcement mechanisms like Intel CET are in place, the first approach cannot be applied because of the shadow stack. The second approach strongly depends on the specific application the attacker wants to exploit, while the last approach is more general and can be applied to any application using the same shared libraries. Being able to find global function pointers in libraries would simplify exploit writing for such applications. For this reason, our goal is to find ① global function pointers calls in the source code of a target library; ② the conditions to reach such calls, giving us 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 thousand to several hundred thousand. Manually searching for global function pointers and all locations where they are called is feasible but not trivial: analyzing a large code base would require considerable time and effort. Even if we could find all function pointers and calls manually, it is challenging to identify the conditions over function parameters and other global variables that would lead the program to the execution of such calls. In fact, some libraries contain functions that are hundreds of source code lines long. Manually keeping track of all the conditions needed to be satisfied to reach a specific code section at runtime would be demanding, time-consuming, and error-prone. Therefore, automating the identification of global function pointer calls is necessary. This would make the whole process faster, more practical, and more reliable, and would provide library developers with an effective way to identify and reduce the attack surface in their code.

5 Untangle

Untangle uses a combination of static analysis, library source code instrumentation, and symbolic execution to provide precise information on how to reach global function pointer calls starting from exported functions of a given C library. This includes information on the constraints on function parameters and global variables that need to be satisfied to reach these calls. The workflow of Untangle includes several components: the *Global Pointers Extractor*, the *Instrumenter*, the *Parser* and the *Executor*, which contains a custom memory model for symbolic execution. The *Global Pointers Extractor* creates a CodeQL database for the library from its source. CodeQL's query language allows specifying precisely the targets of the static analysis: in our case, the targets are global function pointers, their call sites, and library functions that can reach them, along with their signatures. After the creation of the database, the *Global Pointers Extractor* performs queries to identify these targets. The *Instrumenter* then places a call to a uniquely generated *target* function immediately before each identified global function pointer call, and builds a new, instrumented version of the library. The *Parser* performs two different tasks: *struct* parsing and *function signature* parsing. The results of both these tasks are passed to the *Executor* component: the information on function signatures is used by the symbolic execution engine, while the information on structures is used by the custom memory model. The *Executor* uses the instrumented library binaries to evaluate the reachability of identified global function pointer calls, treating the functions inserted by the *Instrumenter* as targets to reach. The actual symbolic execution

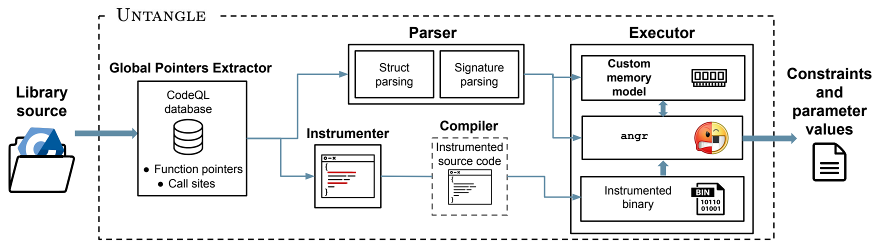

Fig. 1: Architecture overview of Untangle.

is performed by `angr`. We use `angr` because its modular design allows us to easily add new functionality or modify existing behavior. In fact, the *Executor* uses a *Symbolic Memory Model* for `angr`, developed with its plugin system, specifically designed to ease the handling of complex structure pointer parameters. The *Executor* also has a built-in *Automatic Result Validation Mechanism*, that we use to test the correctness of the results of the symbolic execution phase. For an overview of the architecture of our tool, refer to Figure 1.

Global Pointers Extractor. This component performs the static analysis of the source code of the library, which is provided as an input to Untangle. As previously mentioned, we use CodeQL for the static analysis, as it allows us to accurately specify the targets of the analysis through its formal query language.

First, the *Global Pointers Extractor* builds a CodeQL database along with the original library. Then, it runs two queries[2] on the database. The first query performs three simultaneous operations: ① detection of all existing global function pointer variables; ② identification of all the call sites for each detected variable; ③ discovery of potential entry points to reach the call sites. The last operation involves traversing CodeQL's call graph, starting from any function containing one or more call sites, going backward from callee to caller, and listing all non-`static` library functions encountered. We can check whether an identified library function is exported by looking at the exported symbols of the library binaries. The second query detects structure definitions, the fields they are composed of, and their offsets inside the structure. The results of this query are passed to the *Parser*, which will use them to create and manage internal objects representing structure pointers.

Instrumenter. The purpose of the *Instrumenter* is to provide targets for the symbolic execution phase through source code instrumentation. This phase must preserve the original functionality of the library to allow the symbolic execution phase to provide reliable results. For this purpose, the *Instrumenter* inserts a call to a uniquely named dummy target function right before each global function pointer call found by the *Global Pointers Extractor*. This new call has only one artificial side effect that prevents it from being optimized away by the compiler. The instrumented library source code is then re-compiled, and the resulting binaries contain exported symbols referencing the newly inserted target functions. This allows providing `angr` with precise indications on the target addresses.

Parser and Executor. Untangle can find constraints on parameters of exported functions and global variables that need to be satisfied to reach identified global function pointer call sites and then evaluate them to find suitable concrete values. The *Parser* extracts the number and types of parameters from the signature of each function that needs to be symbolically executed, creating symbolic bit-vectors of the appropriate size. For `struct` pointer parameters, the *Parser* also creates the needed `StructPointer` objects as previously discussed. The *Symbolic Memory Model* uses these objects to handle symbolic memory loads and stores to structure pointers during the symbolic execution. To allow the iden-

[2] https://github.com/untangle-tool/untangle/blob/main/untangle/analyzer.py#L82

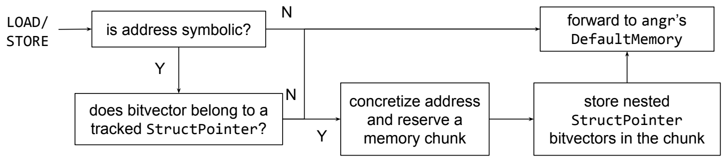
LOAD/ STORE
is address symbolic?
N
Y
does bitvector belong to a tracked StructPointer?
N
Y
concretize address and reserve a memory chunk
store nested StructPointer bitvectors in the chunk
forward to angr's DefaultMemory

Fig. 2: Load/Store handling using Untangle's memory model

tification and evaluation of interesting global variables, the *Executor* transforms writable data sections of the library binary (`.bss` and `.data`) to symbolic bit-vectors to verify later whether any memory regions belonging to these sections were involved in any constraints. This also allows the detection of constraints on the global function pointers themselves. However, these constraints depend on the specific library being tested and need to be evaluated case by case.

Symbolic Memory Model. `Angr`'s default address concretization strategy can cause memory overlaps since it is unaware of variable types and sizes. Therefore, it cannot reserve specific regions of memory for symbolic pointers. A prime example is pointers to `struct` types. To correctly handle `struct` pointers, Untangle extends `angr`'s memory model implementing ad-hoc logic. This logic is summarized in Figure 2. Function arguments that are pointers to known `struct` types, extracted through CodeQL, are recursively parsed into an internal `StructPointer` object, which holds fields' offsets, sizes, and symbolic bit-vectors. During symbolic execution, Untangle keeps track of `StructPointer` objects to handle load/store memory operations involving their addresses. The first load-/store operation through the symbolic bit-vector of a tracked `StructPointer` p concretizes its value to an address determined by a simple bump allocator. At this address, Untangle reserves a chunk of symbolic memory of the needed size to hold the content of the underlying `struct` that p is tracking. Then, Untangle stores the symbolic bit-vectors for any nested `StructPointer` field of p at the correct offset in the chunk. Any subsequent load/store operation to the now-concrete address is then forwarded to `angr`'s default handler. Using this approach recursively, Untangle can also handle *nested* struct pointers.

Automatic Result Validation Mechanism. Untangle is equipped with an automatic result validation mechanism. Validation is performed 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 tested library function. This is not a simple task, and depending on the library, the test program would need to be significantly complex to compile correctly. Using a library function means importing the correct header files, creating variables of the appropriate type and value (which can, in turn, require additional headers for the type definitions), and 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 implemented a simpler automatic verification method that involves the use of `libdl` [5] to dynamically load instrumented

Table 1: List of tested libraries and number of global function pointers found in each library by Untangle.

Library	Estimated lines of source code	Unique global function pointers	Reachable function pointers
libgnutls v3.6.16	422 804	15	14
libasound v1.2.4	94 288	3	2
libxml2 v2.9.10	353 481	8	6
libfuse v3.11	21 568	1	1
libcurl v7.84	152 921	5	5
libnss v2.31	10 568	21	18
libpcre v8.39	107 530	3	3
libbsd v0.11.3	11 316	8	8
Total	1 174 476	64	57

libraries at runtime and **The GNU Debugger** (GDB) to monitor whether identified call sites are reached through automatically inserted breakpoints. The goal of this built-in *Automatic Validation Mechanism* is to avoid false positive results: if execution reaches a breakpoint set at the target call site while running under GDB, the solution must inevitably be correct (it could be trivial, but correct). Therefore, an incorrect solution will never pass validation. This mechanism can, however, yield false negatives: functions for which Untangle found a solution but through which the global function pointer call site is not reached during automatic validation. These are more complex to handle and require manual testing to be identified. Automatic validation consists of the following steps performed after a symbolic execution run that found a satisfiable solution: ① Generate and compile a C program that loads the tested library using `libdl` and calls the target function using parameter values taken from the solution; ② Run the compiled program under GDB, setting a breakpoint on the target function corresponding to the global function pointer call site that needs to be reached; ③ Check whether the breakpoint is reached or not.

6 Experimental Validation

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 our tests was the symbolic execution phase: the success rate of symbolic execution (i.e., what percentage of runs can find and return a solution), the validity of found solutions, and the number 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.

Dataset. We selected top-ranked free, open-source C libraries listed under the "libs" section of the **Debian package Popularity Contest** [3], using the latest version provided by Debian 11 packages. We checked the presence of global func-

Table 2: Number of unique call sites, exported functions, and unique paths to global function pointer calls for each tested library.

Library	Unique call sites	**Exported functions**	Unique paths to call sites
libgnutls v3.6.16	1 338	827	29 817
libasound v1.2.4	383	243	7 739
libxml2 v2.9.10	2 125	225	254 096
libfuse v3.11	110	110	110
libcurl v7.84	271	48	11 238
libnss v2.31	34	15	74
libpcre v8.39	13	12	36
libbsd v0.11.3	8	8	8
Total	4 282	1 488	303 118

tion pointers using CodeQL and analyzed them with Untangle. We manually compiled and checked around 50 libraries, found 8 of them (listed in Table 1), to contain interesting function pointers, and we tested them.

Experimental Setup. As shown in Table 2, the number of unique code paths starting from exported library functions and leading to a global function pointer call can be quite large. Hence, 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. 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 **The GNU C Compiler** (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 (e.g., AVX2, SSE4) was disabled to avoid issues with PyVEX [7, 34], 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 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.

6.1 Symbolic Execution Results

The static analysis results found by Untangle are listed in Table 1 and Table 2. The first table shows the number of *unique* global function pointers found in each library: we ruled out the ones that were not *reachable* through manual analysis. The output of the static analysis contains a list of all global function pointers identified and every library function that can reach a call to one of them. Table 2 presents the number of *exported* functions able to reach a global function pointer

Table 3: Symbolic execution results and validation of successful runs.

Library	Tested functions	Symbolic execution solution		Validation result	
		Found	Not Found	Pass	Fail
libgnutls	827	460 (55.6%)	367 (44.4%)	272 (32.9%)	188 (22.7%)
libasound	243	153 (63.0%)	90 (37.0%)	91 (37.4%)	62 (25.5%)
libxml2	225	139 (61.8%)	86 (38.2%)	60 (26.7%)	79 (35.1%)
libfuse	110	59 (53.6%)	51 (46.4%)	15 (13.6%)	44 (40.0%)
libcurl	48	40 (83.3%)	8 (16.7%)	30 (62.5%)	10 (20.8%)
libnss	15	9 (60.0%)	6 (40.0%)	2 (13.3%)	7 (46.7%)
libpcre	12	9 (75.0%)	3 (25.0%)	6 (50.0%)	3 (25.0%)
libbsd	8	8 (100%)	0	8 (100%)	0
Overall	1488	877 (58.9%)	611 (41.1%)	484 (32.5%)	393 (26.4%)

call. Functions that are not exported cannot be called from a program that uses the library, so they are not interesting for our tests: while testing, we check in the compiled library binary if a function is exported or not and perform symbolic execution only on exported functions. As shown in Table 3, we found a solution for 58.9% (877) of the 1488 total exported library functions analyzed.

As explained in Section 5, UNTANGLE has a built-in validation mechanism, which is necessary to understand which of the solutions found through symbolic execution are correct. Validation results are also summarized in Table 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 *Automatic Validation Method* described before. We can also notice the result of what we explained in Section 2: 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 is a chance that such an instance is a false negative. UNTANGLE will report the solution, but manual testing is needed to understand additional and possibly more complex constraints that were not automatically identified. Finally, looking at runs that did not result in a found solution, we can break down the reason into four categories (shown in Table 4): *Unreachable*, *Timeout*, *Memory*, *Engine Error*.

Unreachable refers to a completed symbolic execution, but the engine determined that none of the identified call sites is reachable. Apart from `angr`'s limitations we discussed in Section 2, this can happen because the constraints leading to call sites are impossible to satisfy. *Timeout* refers to a run halted after exceeding 15 minutes. *Memory* refers to a run halted after exceeding 16GiB of used memory. *Engine error* refers to a run halted because of an internal error of the symbolic execution engine. This happens for multiple reasons, the most common of which are constraints that become too complex (e.g., causing the solver to exceed Python's maximum call stack size) or bugs in the engine code.

Table 4: Break-down of unsuccessful symbolic execution runs

Library	Tested functions	Solution not found	Reason			
			Unreachable	Timeout	Memory	Engine error
libgnutls	827	367 (44.4%)	26 (3.1%)	24 (2.9%)	233 (28.2%)	84 (10.2%)
libasound	243	90 (37.0%)	27 (11.1%)	23 (9.5%)	11 (4.5%)	29 (11.9%)
libxml2	225	86 (38.2%)	10 (4.4%)	12 (5.3%)	41 (18.2%)	23 (10.2%)
libfuse	110	51 (46.4%)	7 (6.4%)	9 (8.2%)	1 (0.9%)	34 (30.9%)
libcurl	48	8 (16.7%)	0	3 (6.2%)	0	5 (10.4%)
libnss	15	6 (40.0%)	0	1 (6.7%)	5 (33.3%)	0
libpcre	12	3 (25.0%)	3 (25.0%)	0	0	0
libbsd	8	0	0	0	0	0
Total	1 488	611 (41.1%)	70 (4.7%)	72 (4.84%)	291 (19.6%)	175 (11.8%)

As we can see from Table 4, the first category is the least common. The most common failure reason is running out of memory. 16GiB is a reasonable amount of RAM; exceeding it indicates accumulating too many symbolic states along the way, which ultimately results in slower running times.

6.2 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. As previously mentioned, we limited each symbolic execution run to 15 minutes and 16 GiB of RAM, and each run exceeding either one of these limits was halted. However, we still collected statistics on halted runs and included them in the computation of the results shown in Table 5. The tests we performed took 1 minute and 36 seconds (on average) for each function that was symbolically executed, and the average memory usage was 4373 MiB. Most of the libraries we analyzed have a much lower average memory usage than the overall average memory usage. Three of the libraries (`libgnutls`, `libxml2`, and `libnss`) have a high average memory usage. This could be due to the complexity of the functions that were symbolically executed: the length of the function, the number of control decisions the function takes, and the number of other functions called inside the analyzed function are all factors that can influence the memory usage of the symbolic execution engine.

7 Impact and Defenses

Impact. In the previous section, we have shown that Untangle can effectively find global function pointers in library code and can also provide reliable information on how to reach a call to one of those pointers. Our work has shown

Table 5: Resource usage statistics collected by Untangle.

Library	Tested functions	Runtime		Average memory usage
		Total	Average	
libgnutls	827	20h 21m	1m 29s	5 252 MiB
libasound	243	7h 52m	1m 56s	856 MiB
libxml2	225	7h 29m	2m 00s	4 889 MiB
libfuse	110	2h 53m	1m 34s	591 MiB
libcurl	48	52m 50s	1m 06s	862 MiB
libnss	15	31m 15s	2m 05s	5 746 MiB
libpcre	12	36s	3s	290 MiB
libbsd	8	8s	1s	318 MiB
Overall	1 488	40h	1m 36s	4 373 MiB

how an attacker can identify global function pointers in library code, which are attack vectors even with Intel CET enabled. To highlight the relevance of the problem addressed in our work, we searched for all Ubuntu 22.04 LTS packages using the libraries we tested. The results of this search are collected in Table 6. The total number of unique packages depending on one or more of the libraries we tested is 1820. The list of all packages installed by default on Ubuntu 22.04 LTS contains 157 of these packages. This means that 8.54% of the default packages on Ubuntu 22.04 LTS (which are 1854 by default) have one or more of the libraries we tested as a dependency. A vulnerability allowing arbitrary writes in one of these packages would allow global function pointer hijacking and enable exploitation in CET-enabled scenarios.

A real-world example of such vulnerability is the heap overflow described in **CVE-2021-43527**[3] and **CVE-2021-43529**[4], affecting Network Security Services (NSS) versions prior to 3.73. This vulnerability affects email clients and PDF viewers that use NSS for signature verification, such as Mozilla Thunderbird, LibreOffice, Evolution, and Evince. NSS is one of the libraries in which we found global function pointer calls during our tests. For this reason, exploiting the heap overflow vulnerability (in any of the programs mentioned above) to perform an arbitrary memory write would enable an attacker to achieve instruction pointer control even in CET-enabled scenarios.

Defenses. As previously mentioned, the static code analysis process implemented in Untangle can help library developers to find global function pointers in their code that can be reached through exported functions. With this information, they can employ appropriate measures to prevent global function pointers from being used as attack vectors for control-flow hijacking exploits.

[3] https://nvd.nist.gov/vuln/detail/CVE-2021-43527
[4] https://nvd.nist.gov/vuln/detail/CVE-2021-43529

Table 6: Number of Ubuntu packages depending on the libraries we used in our tests. The number between parentheses is the number of unique packages (since some of them can have more than one of these libraries as a dependency).

Library	**libgnutls**	**libasound**	**libxml2**	**libfuse**	**libcurl**	**libnss**	**libpcre**	**libbsd**	**Total**
# of packages	252	323	699	31	180	63	209	264	2021 (1820)

This paper demonstrated the relevance of securing function pointers to avoid control-flow hijacking attacks in settings where CFI defenses are in place. Indirect call protection mechanisms already exist in LLVM: *Indirect Function Call Checks* (IFCC) checks the original function pointer's signature against the signature of the function that is actually called through the function pointer. Unfortunately, this mitigation is still not adopted among the major Linux distributions as the most used among them (Ubuntu, Debian, Arch, Fedora) use GCC as the default compiler in their build systems. Consequently, until this countermeasure becomes widespread, the results of Untangle can still be used for exploitation and underline the relevance of indirect call protection mechanisms. Some defense proposals are currently being developed with this goal. **FineIBT** [4] is a software defense proposal for the Linux kernel that builds over CET, adding special instrumentation to the generated binary to enforce the verification of hashes on function prologues whenever these are indirectly called. The hashes are computed over function, and *function pointer* prototypes at compile-time and checked at run-time whenever an indirect call happens.

8 Limitations and Future Work

The main limitations of Untangle come from the tool used for static analysis of source code: CodeQL. As mentioned in Section 5, CodeQL performs its analysis at the source code level, and it does not provide any information about the location of specific instructions or basic blocks in the resulting compiled binaries. First, while it 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 might be 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 variables' assignments and copies throughout the entire code base. CodeQL offers a mechanism to do this through taint analysis but would still be unable to cover all instances. An example is when the address contained in a function pointer is copied using inline assembly, which CodeQL cannot handle. Another limitation of Untangle is the way instrumentation is performed. Depending on how the library is written, it is not always possible to place a function call before the identified function pointer call without changing the original semantics of the program. In fact, in specific situations where complex macros are involved, we cannot apply our instrumentation method as-is: the only way to analyze such cases is to manually expand every instance of the macro before instrumenting it (which was the case with **gnutls** in our tests). The information provided by CodeQL makes the location of global function pointer call sites only identifiable at the source code level. Extracting 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** [6] 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 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 provide those directly to `angr` as a target for symbolic execution. Because of its design, Untangle needs the library's source code 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** [38], which enable static analysis of binary executables, could be leveraged along with heuristics to identify which call sites to consider as global function pointer calls. Searching for all the indirect calls in a binary and evaluating if they can be hijacked could be an extension of what Untangle already does and could be interesting to investigate. However, this task is challenging as it would be computationally expensive to perform through symbolic execution.

9 Conclusions

This work provides an automated methodology for finding global function pointers whose calls are reachable through exported C library functions, along with all the constraints that need to be satisfied to reach them. The approach we present 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 to find constraints on function parameters and global variables that need to be satisfied to reach such calls. We present Untangle, a tool that implements this approach to assist 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 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 Intel CET enabled, 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.

Bibliography

[1] angr. https://angr.io/

[2] CodeQL. https://codeql.github.com/

[3] Debian popularity contest. https://popcon.debian.org/main/index.html

[4] Fine-grained forward CFI on top of intel CET / IBT. https://www.openwall.com/lists/kernel-hardening/2021/02/11/1

[5] Linux standard base specification: Interface definitions for libdl. https://refspecs.linuxbase.org/LSB_3.0.0/LSB-generic/LSB-generic/libdlman.html

[6] The LLVM compiler infrastructure. https://llvm.org/

[7] PyVEX. https://github.com/angr/pyvex

[8] Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control-flow integrity principles, implementations, and applications. ACM Trans. Inf. Syst. Secur. **13**(1), 4:1–4:40 (2009). https://doi.org/10.1145/1609956.1609960

[9] Bhatkar, S., DuVarney, D.C., Sekar, R.: Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In: Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003. USENIX Association (2003)

[10] Bletsch, T.K., Jiang, X., Freeh, V.W.: Mitigating code-reuse attacks with control-flow locking. In: Twenty-Seventh Annual Computer Security Applications Conference, ACSAC 2011, Orlando, FL, USA, 5-9 December 2011. pp. 353–362. ACM (2011). https://doi.org/10.1145/2076732.2076783

[11] Bletsch, T.K., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, Hong Kong, China, March 22-24, 2011. pp. 30–40. ACM (2011). https://doi.org/10.1145/1966913.1966919

[12] Borzacchiello, L., Coppa, E., D'Elia, D.C., Demetrescu, C.: Memory models in symbolic execution: key ideas and new thoughts. Softw. Test. Verification Reliab. **29**(8) (2019). https://doi.org/10.1002/stvr.1722

[13] Buchanan, E., Roemer, R., Savage, S., Shacham, H.: Return-oriented programming: Exploitation without code injection. Black Hat **8** (2008)

[14] Burow, N., Carr, S.A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., Payer, M.: Control-flow integrity: Precision, security, and performance. ACM Comput. Surv. **50**(1), 16:1–16:33 (2017). https://doi.org/10.1145/3054924

[15] Carlini, N., Wagner, D.A.: ROP is still dangerous: Breaking modern defenses. In: Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014. pp. 385–399. USENIX Association (2014)

[16] Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: Proceedings of the 17th ACM Conference on Computer and Communications Se-

curity, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010. pp. 559–572. ACM (2010). https://doi.org/10.1145/1866307.1866370

[17] Chen, S., Xu, J., Sezer, E.C.: Non-control-data attacks are realistic threats. In: Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005. USENIX Association (2005)

[18] Cheng, Y., Zhou, Z., Yu, M., Ding, X., Deng, R.H.: Ropecker: A generic and practical approach for defending against ROP attacks (2014)

[19] Dang, T.H.Y., Maniatis, P., Wagner, D.A.: The performance cost of shadow stacks and stack canaries. In: Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015. pp. 555–566. ACM (2015). https://doi.org/10.1145/2714576.2714635

[20] Davi, L., Sadeghi, A., Winandy, M.: Ropdefender: a detection tool to defend against return-oriented programming attacks. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, Hong Kong, China, March 22-24, 2011. pp. 40–51. ACM (2011). https://doi.org/10.1145/1966913.1966920

[21] Homescu, A., Stewart, M., Larsen, P., Brunthaler, S., Franz, M.: Microgadgets: Size does matter in turing-complete return-oriented programming. In: 6th USENIX Workshop on Offensive Technologies, WOOT'12, August 6-7, 2012, Bellevue, WA, USA, Proceedings. pp. 64–76. USENIX Association (2012)

[22] Hu, H., Chua, Z.L., Adrian, S., Saxena, P., Liang, Z.: Automatic generation of data-oriented exploits. In: 24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015. pp. 177–192. USENIX Association (2015)

[23] Hu, H., Shinde, S., Adrian, S., Chua, Z.L., Saxena, P., Liang, Z.: Data-oriented programming: On the expressiveness of non-control data attacks. In: IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016. pp. 969–986. IEEE Computer Society (2016). https://doi.org/10.1109/SP.2016.62

[24] Ispoglou, K.K., AlBassam, B., Jaeger, T., Payer, M.: Block oriented programming: Automating data-only attacks. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018. pp. 1868–1882. ACM (2018). https://doi.org/10.1145/3243734.3243739

[25] King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976). https://doi.org/10.1145/360248.360252

[26] Niu, B., Tan, G.: Modular control-flow integrity. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014. pp. 577–587. ACM (2014). https://doi.org/10.1145/2594291.2594295

[27] Pappas, V., Polychronakis, M., Keromytis, A.D.: Transparent ROP exploit mitigation using indirect branch tracing. In: Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013. pp. 447–462. USENIX Association (2013)

[28] Prandini, M., Ramilli, M.: Return-oriented programming. IEEE Secur. Priv. **10**(6), 84–87 (2012). https://doi.org/10.1109/MSP.2012.152

[29] Roemer, R., Buchanan, E., Shacham, H., Savage, S.: Return-oriented programming: Systems, languages, and applications. ACM Trans. Inf. Syst. Secur. **15**(1), 2:1–2:34 (2012). https://doi.org/10.1145/2133375.2133377

[30] Sadeghi, A., Niksefat, S., Rostamipour, M.: Pure-call oriented programming (PCOP): chaining the gadgets using call instructions. J. Comput. Virol. Hacking Tech. **14**(2), 139–156 (2018). https://doi.org/10.1007/s11416-017-0299-1

[31] Schuster, F., Tendyck, T., Pewny, J., Maaß, A., Steegmanns, M., Contag, M., Holz, T.: Evaluating the effectiveness of current anti-rop defenses. In: Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8688, pp. 88–108. Springer (2014). https://doi.org/10.1007/978-3-319-11379-1_5

[32] Shacham, H., Page, M., Pfaff, B., Goh, E., Modadugu, N., Boneh, D.: On the effectiveness of address-space randomization. In: Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, DC, USA, October 25-29, 2004. pp. 298–307. ACM (2004). https://doi.org/10.1145/1030083.1030124

[33] Shanbhogue, V., Gupta, D., Sahita, R.: Security analysis of processor instruction set architecture for enforcing control-flow integrity. In: Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy, HASP@ISCA 2019, June 23, 2019. pp. 8:1–8:11. ACM (2019). https://doi.org/10.1145/3337167.3337175

[34] Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., Vigna, G.: Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware (2015)

[35] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Krügel, C., Vigna, G.: SOK: (state of) the art of war: Offensive techniques in binary analysis. In: IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016. pp. 138–157. IEEE Computer Society (2016). https://doi.org/10.1109/SP.2016.17

[36] Szekeres, L., Payer, M., Wei, T., Song, D.: Sok: Eternal war in memory. In: 2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013. pp. 48–62. IEEE Computer Society (2013). https://doi.org/10.1109/SP.2013.13

[37] Wang, F., Shoshitaishvili, Y.: Angr - the next generation of binary analysis. In: IEEE Cybersecurity Development, SecDev 2017, Cambridge, MA, USA, September 24-26, 2017. pp. 8–9. IEEE Computer Society (2017). https://doi.org/10.1109/SecDev.2017.14

[38] Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014. pp. 590–604. IEEE Computer Society (2014). https://doi.org/10.1109/SP.2014.44