Automatic Exploit Generation for Cross-Language Attacks

by

Yosef E Mihretie
S.B. in Electrical Engineering and Computer Science
Massachusetts Institute of Technology (2021)

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2022

Author .....
Department of Electrical Engineering and Computer Science
Aug 18, 2022

Certified by .....
Nathan Burow
Member of Technical Staff
Thesis Supervisor

Certified by .....
Hamed Okhravi
Senior Member of Technical Staff
Thesis Supervisor

Certified by .....
Howard Shrobe
Principal Researcher
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Automatic Exploit Generation for Cross-Language Attacks

by

Yosef E Mihretie

Submitted to the Department of Electrical Engineering and Computer Science
on Aug 18, 2022, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Memory corruption is an essential component of most computer exploits. At the same time, a significant portion of legacy system software is written in C/C++, which are known to be memory-unsafe. This has led to an arms race between attackers devising ever clever ways to execute memory corruption and developers engineering mitigation techniques to either prevent or raise the alarm when memory is corrupted. This has come to be known as "The Eternal War in Memory". Recently, however, software programmers have shifted to using programming languages that are memory-safe by design like Go and Rust. These languages are specially favorable because they provide an easy interface that allows them to interact with the widely established C/C++ based infrastructure. Underlying this design approach is the assumption that replacing parts of a largely memory-unsafe software program with memory safe code will raise the overall security of the program. Recent work has however showed this assumption is flawed. In fact, mixing sections with different threat models into one program can lead to attacks that would not have been possible in the two sections individually. These attacks are called Cross-Language Attacks (CLA). On the other hand, analyzing large binary programs to construct CLA exploits is a tedious process. In this thesis, we present ACLEG which automatically generates CLA for the case of double-free exploits. ACLEG can help researchers and engineers understand the extent of CLA vulnerabilities in commercially deployed software programs. Moreover, it can help find bugs in software programs before they are deployed as part of the debugging toolset.

Thesis Supervisor: Nathan Burow
Title: Member of Technical Staff

Thesis Supervisor: Hamed Okhravi
Title: Senior Member of Technical Staff

Thesis Supervisor: Howard Shrobe
Title: Principal Researcher

Acknowledgments

Special thanks go to my advisors Dr. Nathan Burow, Dr. Hamed Okhravi and Sam Mergendahl for first giving me the opportunity to work on this exciting project and then for the patience and accommodation they have extended to me in some of the most stressful moments of my life. My family, friends, teachers since childhood and all those I have met in this life of mine are sincerely a part of me. I could not have gotten thus far without your continued motivation, help and love. Cheers!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

C/C++ leave the responsibility of memory management to the programmer, leading to a plethora of memory corruption vulnerabilities. An attacker can take over the control flow of a program if they can overwrite control flow data [19]. Corruption of non-control flow data can also lead to exploits that alter the behavior of the program [32]. Despite these vulnerabilities, however, due to the prevalence of software already built in C/C++ and their performance benefits, they continue to dominate the systems programming world.

Over the years, multiple mitigation techniques have been proposed to strengthen C/C++ [67, 16, 55, 8]. Some of the proposed solutions require unrealistic performance costs [65], removing the high performance advantages of writing programs in of C/C++. Other mitigation techniques have been shown to be inadequate [58, 26]. DEP can be bypassed by code reuse attacks [19, 49]. ASLR can be bypassed by information leaks [58]. Precise enforcement of CFI has a large overhead and the less relaxed versions of CFI that are deployed have been shown to allow enough transfers for building a malicious payload [26]. This arms race between defense mechanisms and even more sophisticated attacks has been termed the eternal war in memory [65].

Another approach to defending against memory corruption attacks is to enforce memory safety by design. Languages like Go [3] and Rust [43] provide both spatial and temporal memory safety while keeping the performance benefits of C/C++. A common approach is to incrementally migrate from C/C++ to Go or Rust, a de-

ployment decision that has been enabled by the interfaces Rust and Go provide for compatibility with C/C++. Developers can outsource some parts of their software to Go or Rust and compile everything into one mixed binary that runs in one address space.

The underlying assumption in this approach is that software written in an unsafe language is made more safe by writing some of its components in a memory-safe language. Unfortunately, however, recent studies have shown that this assumption is incorrect [44, 54]. We consider mixed binaries where the C/C++ side deploys many of the famous protections like CFI, DEP, Stack Canaries and ASLR. What the studies have shown is that the safe side of these binaries enables easy exploit of the unsafe but hardened and otherwise not trivially exploitable part of the software. In other words, exporting some functionalities of a software written in an unsafe language to a safe language degrades the overall security of the system. These studies demonstrate a new attack vector - Cross-Language Attacks or CLA for short [44].

Finding an exploitable vulnerability of any kind, including CLA, is not a trivial task. It requires reasoning about a program, understanding the underlying assumptions, deployed protections and finding inputs that take the program to a point of unexpected behavior. Done manually, this can be an exhausting process. Specifically, software programmers need to reason about their program and it is hard to prove safety manually. To this end, recent progress has been in the field of automatically generating these exploits. This is specially important in complex programs in binary format, which is the case for commercial programs. To understand the extent to which commercially deployed programs are vulnerable to CLAs, it is critical to develop automatic exploit generation capabilities targeted at discovering Cross-Language vulnerabilities.

Automatic Exploit Generation, AEG for short, is a process of analyzing a program and the environment the program is run in to automatically generate exploits against it. Fuzzing [45] is one of the earlier techniques that AEG tools depend on. In fuzzing, random inputs are generated and fed into a program until a desired state is reached. The inputs that engender these desired states of exploitation then be-

come the payload. Since fuzzing is a random process with no quantifiable limits to its performance, researchers and developers usually rely on alternative methods like symbolic execution [61]. Symbolic execution represents the inputs to a program as variables and reasons how they evolve in the course of execution. Symbolic execution engineers create multiple lines of execution at branch statements while keeping satisfiability conditions. If a desired state is reached this way, the satisfiability conditions are solved and a payload is constructed from these solutions. End-to-end exploit generation tools in the literature include works like AEG [10] and BOPC [35]. AEG constructs end-to-end control-flow hijacks while BOPC generates Data-oriented programming exploits for an arbitrary set of instructions. Both of these tools rely on symbolic executions.

In this thesis, we implemented an Automatic Exploit Generation system that analyzes software in binary formats to construct CLA exploits. Our work is built on top of the binary analysis tool Angr. We target big, commercial software programs like Firefox and Fuchsia. A summary of our contributions:

- A generic novel three step Automatic Exploit Generation method that scales to complex programs with millions of lines of code
- A novel symbolic execution pruning method that addresses the path explosion problem built on top of Angr's inbuilt symbolic execution engine
- A complete CLA profiling of Firefox and Fuchsia software programs to help guide future work

Chapter 2

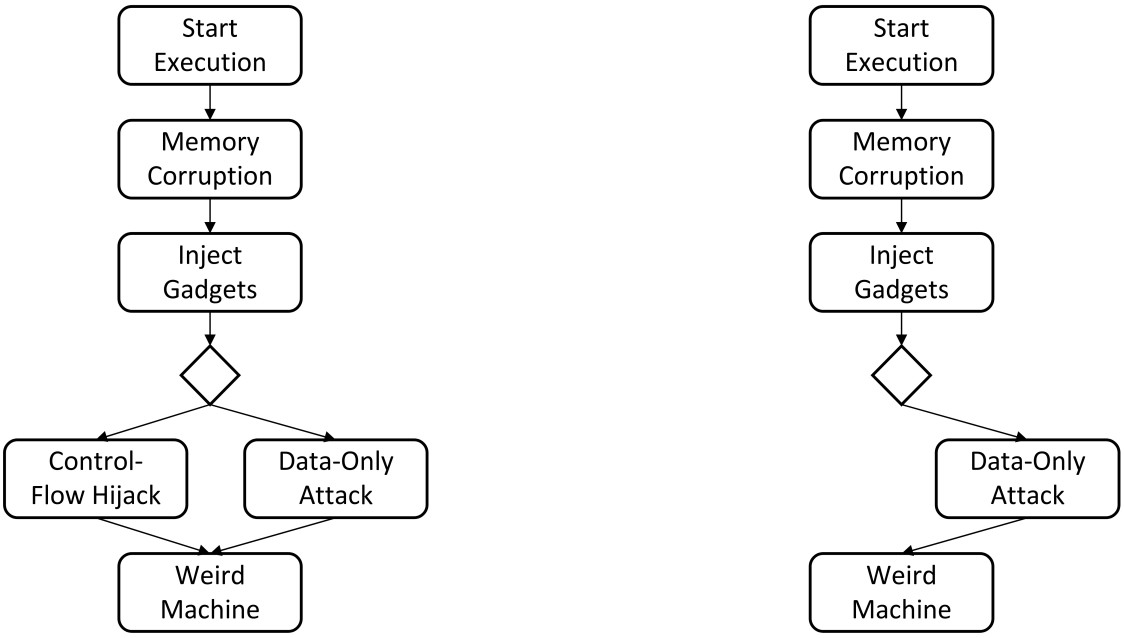
Background

C/C++ have dominated the systems programming landscape for decades now. This is due to the significant performance benefits these low-level programming languages offer. However, these two languages by design vulnerable to memory-corruption attacks [29]. Despite efforts to mitigate this vulnerability, attacks have designed ever more refined attacks, leading to what has been termed as the "Eternal war in memory". Research in this area has also been expanded to designing tools for automatically generating these exploits so that they can be better understood and preemptively fixed.

Recently, memory-safe programming languages like Go and Rust have been getting more traction in the systems programming industry. These two languages have performances that are comparable to that of C/C++ and have interfaces that allow them to integrate with legacy code written in C/C++. Thus, developers have been slowly replacing parts of existing code to create what are called Cross-Language software programs. This however has been shown to open a new class of exploits termed Cross-Language Attacks(CLAs) [44].

2.1 War in memory

Memory corruption occurs when an attacker is able to read from or write to addresses in violation of the memory access rules. A typical memory corruption attack is demonstrated in figure 2-1a below. The attacker first steers execution towards a point



(a) C/C++ without mitigations

(b) C/C++ with CFI deployed

Figure 2-1: Threat models for C without and with CFI

where there is memory vulnerability and then uses the vulnerability for corrupting a memory location. This corruption can be used to modify control data and takeover control flow or non-control data and execute an undesired computation. Either way, a weird machine is achieved.

Memory corruptions can be classified into two major classes: temporal and spatial [65]. Temporal memory corruption occurs when an attacker makes use of a pointer that points to a deallocated space. Two common examples of this class are dangling pointers and double frees. Spatial memory corruption occurs when an attacker uses a pointer beyond the bounds of its referent. Buffer overflows are a classic example of spatial memory corruption. Either way an attacker can use a memory corruption to inject code, overwrite control data and takeover control flow, overwrite non-control data and achieve an execution that is not intended by the programmers, leak information and more.

Since C/C++ by design can not provide memory safety, mitigation techniques focus on controlling damage after a memory corruption occurs. Examples of these

mitigation techniques that are widely deployed are Data Execution Prevention (DEP), Stack Canaries, Address Space Randomization (ASLR) and Control Flow Integrity (CFI). Figure 2-1b demonstrates the threat model for C/C++ deployed with CFI. As can be seen, the attacker can no longer takeover control flow.

Data Execution Prevention (DEP) is a defense mechanism that marks the pages where the program can write, like heap and stack, not executable. This defense prevents code injection attacks where the attacker uses a memory corruption vulnerability to send code snippets as data and execute them. However DEP can be bypassed by Return Oriented Programming(ROP) [19], where an attacker chains together gadgets within the existing program space instead of injecting them. Studies have shown that commonly available libraries like libc provide enough gadgets for Turing completeness [66] and there are tools that automate the gadget discovery process [28]. An attacker could chain these gadgets to make a non-executable page executable. In Linux, for instance, this is achieved by calling *mprotect* [4].

Stack Canaries provide protection against stack smashing buffer overflow attacks, where the attacker takes over a system by modifying the return address of a function via an overflow of a buffer on the stack that is adjacent to the return address [53]. A stack canary is a security code that is inserted between the local variables and meta information of a stack frame [67]. Once the function returns, this stack canary is checked to make sure it was not overwritten. If it was, then that is a sign that an overflow happened and an exception was triggered. Stack Canaries are bypassed if the attacker can guess the canary value or the value can be leaked (by using a dangling pointer to read it for instance) or by a direct write using a pointer [13].

ROP [19] works by chaining together code snippets that are already part of the code-base. This requires knowing where the base address of each code source is. **Address Space Layout Randomization** (ASLR) prevents code-reuse attacks by making the base addresses of the heap, stack, shared libraries, the executable, PLT etc unpredictable. However there are many ways to bypass ASLR. The first method is to simply brute-force until the right base addresses are found. A more practical

approach is to leak information about the base addresses via another attack [58].

The ultimate goal of most memory corruption attacks is to takeover the control flow of the system. This usually entails making calls or jumping to locations that were not part of the original code. CFI protects against these by statically generating a control flow graph (CFG) at compile time and allowing jumps or function calls only when they conform to the generated CFG. Due to the large overheads of enforcing precise CFI, CFI is usually enforced on a coarse-grained level. Studies have shown that sufficiently complex hijacks can be performed even while obeying the CFG [26]. Even if CFI is completely enforced, Data Oriented Programming (DOP) [33] have been shown to enable takeover by an attacker.

2.2 Memory-safe programming languages

As shown above, almost every practical hardening mechanism has been proved to not be bullet proof. This led to the development of languages that provide memory safety inherently while keeping a performance comparable to C/C++. These languages focus on making memory corruptions impossible in the first places, rendering anything that comes after irrelant. This is demonstrated in figure 2-2. For enhanced performance, these languages rely on performing most of their memory-safety checks at compile time and minimizing runtime interference. In this section, we discuss how Rust and Go manage to do so.

Rust is an open-source programming language developed by Mozilla. Rust enforces spatial security by performing bounds checks statically at compile time or dynamically with small snippets inserted in the binary. It also has a strong type system that prevents arbitrary casts which are known to cause many security bugs. Additionally, Rust enforces an ownership policy where every value is owned by other variables. Rust allows either only one variable owner with a mutable reference to a value or multiple variables with immutable references. This makes it easy to claim memory not used anymore easily and enforces temporal safety. Rust's Foreign Function Interface (FFI) [2] allows it to interact with C/C++, where the two can be

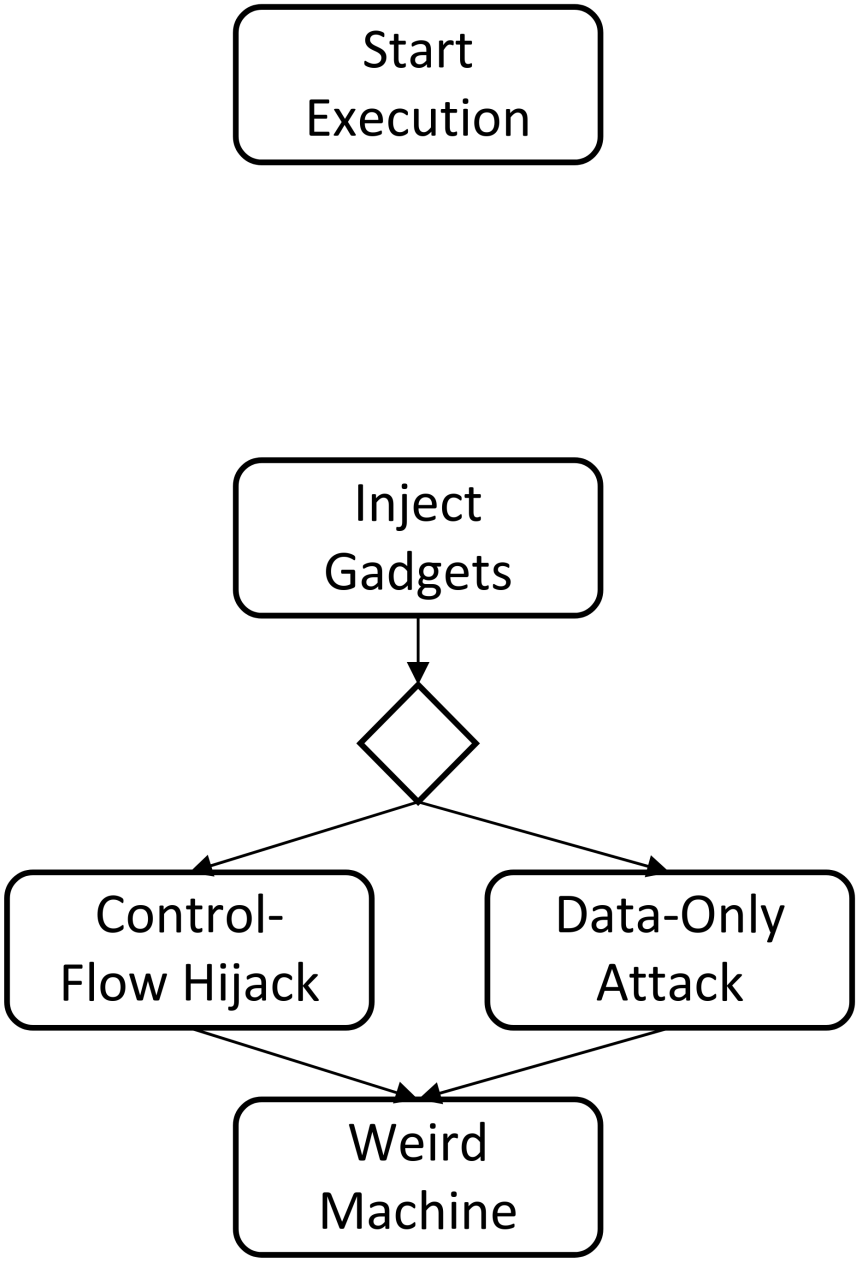


Figure 2-2: Memory-safe programming languages remove the first step in the attack chain

compiled into a single mixed binary that can be run in one address space.

Go is also an open-source programming language that is sponsored by Google. Similar to Rust, it enforces spatial safety via dynamic or static bounds checks. It is also statically and strongly typed, preventing casting bugs. Go uses a garbage collector to enforce temporal safety. Similarly to Rust, Go provides CGO[1], an interface that allows it to interact with C/C++ code.

2.3 Cross-Language Attacks

As stated above, a popular design in many programs is to slowly replace parts of legacy code written in memory-unsafe languages like C/C++ with memory-safe languages like Rust/Go. This is made possible by the interfaces Rust/Go provide for integrating with C/C++. The assumption in this design is that using memory-safe languages to rewrite parts of a program largely written in memory-unsafe languages will strengthen the overall security of the program. This however has recently been shown to not be the case. Cross-Language Attack(CLA) [44] is a new class of attacks that leverages the mismatch of threat models in programs written with multiple languages to undermine the security of both sides of the program.

The CLA threat model is a union of the threat models of its constituent languages. This is shown in figure 2-3 above. An attacker can start in one of the languages, steer execution to the language transfer point and proceed its attack in the other language landscape. Effectively, combining two relatively safe landscapes results in an unsafe system. CLA does not even require one side to explicitly call the other; if the safe side is run parallel to the unsafe side, a memory corruption on the unsafe side can be used to corrupt the memory a safe side relies on.

An example of an attack made possible by the CLA threat model is bounds check bypass. For vectors in Rust and slices in Go, the system stores a length variable against which every access is checked for bounds. This can be bypassed if a memory corruption in the C/C++ side of the mixed binary can be used to overwrite this length variable to a desired value. Another example is lifetime bypass attack. Since

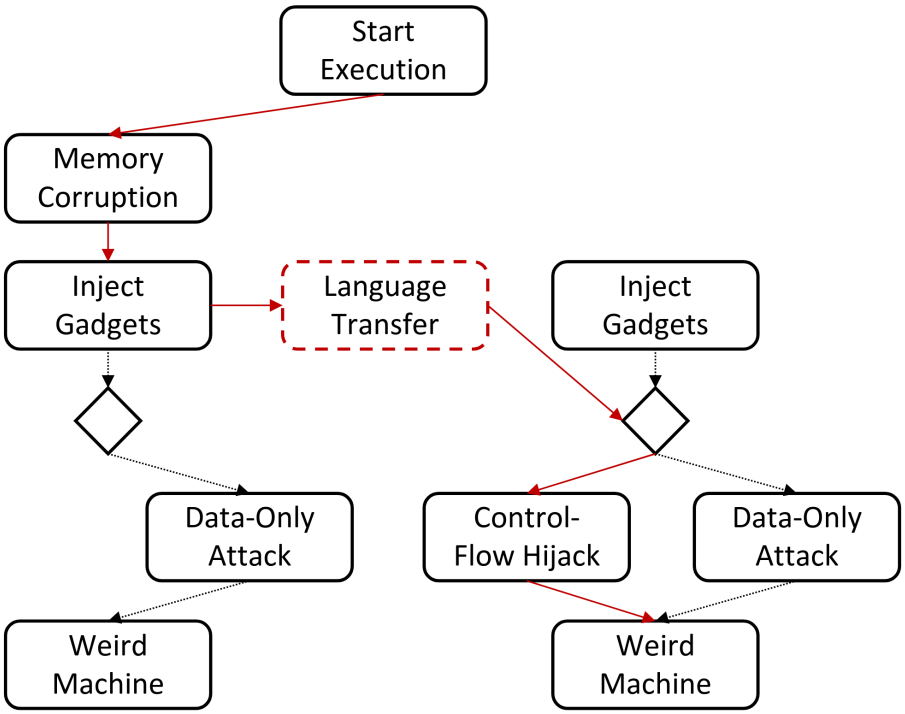


Figure 2-3: CLA exploits the mismatch of threat models to bypass security guarantees in both sides

C/C++ and Rust use the same heap management system, if the C/C++ side is compromised to execute a free on a Rust allocated address, then that bypasses the lifetime checks that the Rust side would ultimately perform. This can lead to use after free and double free attacks. C/C++ hardening bypass is another interesting area. Some of the hardening techniques for mitigating C/C++ software can be made easy to bypass via a CLA. For instance, CFI can be bypassed by using a memory corruption in C/C++ to make arbitrary jumps or function calls in Rust.

2.4 Exploit generation

Commercial software programs are complex and usually in binary format. Manually reasoning about such programs and finding exploits is a tedious process. Moreover, proving the security of a program is even more difficult as developers would need to think about every possible exploit. Thus, developers rely on Automatic Exploit

Generation (AEG) [10, 35], bopc techniques. AEG is the process of automatically finding vulnerabilities in a program and the necessary conditions for exploiting them. AEG tools are important for verifying the security of software programs.

In the most general case, AEG tools take an input binary and a description of what the attacker wants to achieve. A description of what an attacker wants can be a simple sequence of instructions as done in the BOPC research [35]. The AEG tool outputs what input should be given to the program and how in order to achieve the desired attack. For instance if there is an HTTP server program that contains a buffer overflow vulnerability and the attacker wants to launch a shell, an AEG tool would ideally discover the vulnerability and report what inputs to send to the server to launch a shell. AEG tools can also be specifically tailored to automatize a specific of the attack chain. For instance, they can be limited to discovering the vulnerabilities which the attacker can manually analyze to construct an end-to-end attack.

A typical AEG proceeds by first finding a vulnerability in the source code by exploring different execution paths. This is usually done via symbolic execution [61, 10]. Working back from the point of vulnerability, a set of constraints on the inputs to the program that trigger the execution path to the point of vulnerability is generated. Then usually a dynamic analysis is performed at the point of vulnerability to collect information on the memory layout. This can then be used to figure out another set of constraints to on the inputs to achieve the execution state desired by the attacker. A union of the two sets of constraints is solved to figure out the inputs that can enable the attack.

2.5 Symbolic Execution

In the process of execution, there are several conditional statements, indirect calls, loops etc that are dependent on the input data. In real life execution where the input data is known, only a single path gets executed at a single time. To explore every possible path the program could possibly take, and thus help exhaustively search for exploitable conditions, we would need to feed the program as many different inputs

as there are paths. That is an exponential number of inputs. This is an extremely efficient way to search for exploitable conditions.

Symbolic execution is an alternative mode of analyzing a program for exploitation. In symbolic execution, all input data to the program and all other data that is dependent on input data are taken as indeterminate variables. These variables are called symbolic. We are interested in solving what these variables need to be set to achieve a certain behavior in the program. The symbolic execution engine reasons about the program line by line. For expressions that operate on non-symbolic data, it performs the operations regularly. For expressions that operate on one or more symbolic values and write a value back, it simply gets the input expressions and defines the value to be written back as some symbolic expression that is a function of the inputs. For conditional expressions that depend on symbolic data, the symbolic execution engine divides execution into two branches. One follows the path where the condition is true and one follows the paths where the condition is false. To both states, the engine attaches constraints on the symbolic data that would be necessary for execution to follow that path. If the one of the states in the engine reach a desired state, the execution can stop and the constraints can be solved to determined what input triggers that execution line. This can be done for multiple different conditions. Once a certain behavior in the program is observed, the engine will solve the collection of constraints that were needed to get to the desired behavior and it solves them to assign concrete values to the symbolic input. Running the program with the symbolic input set to the concrete values found will lead the program to the desired behavior.

The major issue in symbolic execution is what is called the path explosion problem. As stated above, there is an exponential number of execution paths in a program. If there are n conditional branches in a piece of software, then at each one of them the program creates two lines of execution for a total of $O(2^n)$. The symbolic execution engine would need to manage a memory model, operating system model, file system model for each one of those execution threads. Thus a practical use of symbolic execution in a big commercial software requires a careful pruning of the execution

tree.

2.6 Binary Analysis

Most commercial software is distributed in binary format. In many instances, attackers do not have access to the source code. In that case, all analysis for automatic exploit generation has to be conducted at the level of the binary code. This makes AEG even more harder as at the binary level many details of the program can be obfuscated.

The process of taking reasoning about the behavior of a program in binary format is called binary analysis. Today, there are many tools that abstract away many of the steps involved in analysing a binary and export a defined interface programmers and researchers can work on. One among these tools and the one we used in our work is Angr.

Angr is an open-source binary analysis tool made by the Mechanical Phish team for the DARPA Grand Cyber challenge. Angr uses the CLE library to handle loading a binary and its associated libraries. A binary can be loaded alone, along with all its library dependencies or a select few of them. This flexibility allows programmers to balance their analyses to the computing resources they have. After loadin a binary, Angr lifts the binary program to the VEX intermediate level representation, on which its analyses are performed. This allows Angr to be architecture independent. Lifting a binary to VEX is handled by the PyVEX framework. A wrapper around Z3 called Claripy is also used to handle constraints solving for symbolic data.

One of the most basic analyses Angr supports is Control Flow Graph (CFG) recovery. Angr supports two forms of CFG recovery: CFGFast and *CFGEmulated*. *CFGFast* is generated quickly by statically reasoning about the binary. However it is at times inaccurate and incomplete. On the other hand, *CFGEmulated* is constructed after symbolically executing the entirety of hte program until complete covering is achieved and then using that data to infer the CFG. It is highly accurate, but is infeasible even for a small program.

Another relevant feature of Angr is its symbolic execution engine. Angr allows creating program states either from the program entry point or from the entry point of specific functions. In the later case, all the arguments to the function or other global data it accesses are taken as symbolic. In either case, the program states are wrapper with what are called simulation managers that manage a collection of states through a symbolic execution. The simulation managers handle advancing each step forward by a basic block, removing states when they become unsatisfiable and adding multiple states with different constraints when branches are reached. The simulation manager instances also take find and avoid conditions which are used to check the status of every state and stop execution once a find or avoid state is reached. In either case, the states that matches the find and avoid states are classified as such and returned to the user. We make a liberal use of these features in our symbolic execution algorithms.

Additionally, Angr provided a Reaching Definition Analysis (RDA) feature. RDA is an analysis technique which statically determines which definitions may reach a given point in a piece of code. Angr's RDA takes a CFG graph representing the piece of code we would like to analyze, a point in the code where we would like to observe and then generates all the definitions that reach that point in the code. This can come in handy when one is trying to determine for instance what the arguments of a function are.

Chapter 3

Design

ACLEG is designed targeting complex, commercial software programs in binary format. The goal of ACLEG is to understand the prevalence of CLA vulnerabilities as well as provide a tool for programmers that can be used to test for them. We assume a realistic threat model. The attacker does not have access to the source code. This is the case for most commercial software programs which are often distributed in binary format. The program is composed of code sections with two major different security assumptions: one where memory safety is assumed and thus no memory corruption mitigation techniques are deployed, another where memory is assumed unsafe and thus major defense mechanisms are deployed.

As a first proof of concept for our technique, ACLEG generates only double free exploits among the many different cross-language attacks that are possible. Safe languages like Rust and GO deploy garbage collection for memory management while unsafe languages like C/C++ leave that to the programmer. The vulnerability our tool looks for is overwriting a point the C/C++ side deallocated with a live pointer to an unprotected, i.e. Rust/GO, allocated heap memory. This will cause a double free vulnerability when Rust/GO's garbage collected tries to deallocated an already deallocated memory later.

ACLEG is composed of three major phases. The first phase generates candidate execution chains from an allocating protected function to a deallocating unprotected function. The second phase leverages symbolic execution to determine if these chains

are satisfiable, i.e. can be executed in the order specified. Given a set of satisfiable chains, the last phase extracts the necessary information to carry out a full cross-language double free exploit.

3.1 Goals

Manually generating exploits is a tedious process. An attacker needs to reason about all aspects of a program and an exponential number of different execution paths along with uncountable numbers of possible data values possible in each execution path. The difficulty is further exacerbated by the fact that the would be attacker may not have access to the source code and thus have to work on the binary, which is often the case with commercial programs. The difficulty with manually constructing exploits for a binary program is also heightened by the fact that binaries are built for specific architectures and the attacker needs to be comfortable across different architectures to be able to successfully construct exploits for these binaries.

Our tool is built for analyzing any binary and automatically generating double free exploits among the many different cross-language attacks possible.

- raise awareness about the prevalence and seriousness of cross-language attacks
- be able to automatically generate exploits in cross-language software
- assist software engineers better test their code against CLA attacks

3.2 Threat model

Our tool works on program in a binary format. The program is assumed to contain code that has different protections deployed. We presume this is done so at the level of functions, i.e. we presume different functions have different mitigation techniques deployed. This typically arises when different programming languages with different threat models are used to compose the program. We especially presume two different mitigation classes deployed. The first type, which heretofore we will call protected, is

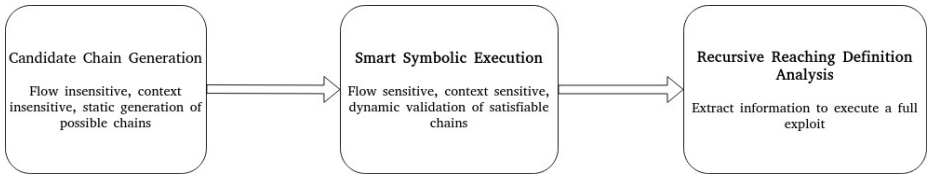
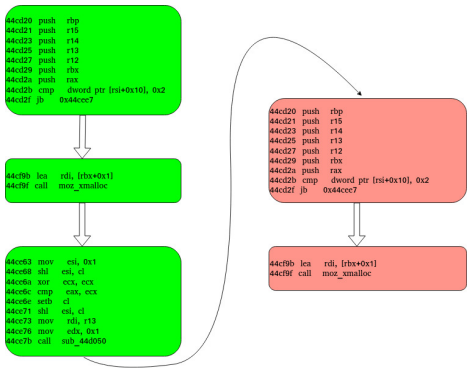


Figure 3-1: The three phases of ACLEG's exploit generation process

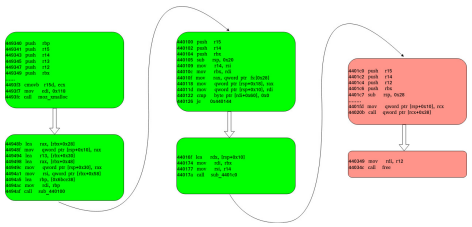
where stack canaries and control flow integrity are deployed. This is usually the case for software written in C/C++. The second class, which we call unprotected, does not have these techniques deployed but is inherently memory-safe. This is the case for code written in Rust and Go. Additionally, we presume Address Space Layout Randomization is deployed. We assume that the attacker has access to a vulnerability in the protected section of the code that allows him to execute an arbitrary memory read and write and we would like to escalate this vulnerability to allow exploiting the unprotected but memory-safe section of the code. We use double frees as the initial target for ACLEG.

3.3 Overview of design

Our exploit generation process is composed of three distinct phases as noted in fig. 3-1: candidate chain generation, flow-sensitive chain selection via symbolic execution and finally recursive reaching definition for identifying memory locations holding sensitive information. The first phase statically generates candidate sequence of basic blocks that would need to be executed in the order defined for a Cross-Language double free exploit. The sequence of basic blocks that emerge from this step are flow and context insensitive: there is no guarantee there is an input that would trigger their execution in the order specified and we have no knowledge of the execution state at any point in the sequence. The purpose of the first phase is to make the flow and context sensitivity adding phase of exploit generation computationally tractable. The second phase is a novel symbolic execution based strategy that uses information generated in the first phase to explore the sequences of basic blocks and find what inputs are required to achieve their execution in the order specified, if at all. Once



(a) An example of a chain spanning two functions



(b) An example of a chain spanning three function

Figure 3-2: Candidate chains taken from the firefox binary file. Green basic blocks are from unprotected functions while red basic blocks are from protected functions.

this phase is done, we know what sequences of blocks can be executed and how the memory needs to be setup to achieve that execution. Additionally, we know the exact sequence of basic blocks that get executed as a result. To fully realize a double free exploit, we need to know what reads and writes are required. This is achieved by the third phase which is a recursive reaching definition analysis that learns what stack locations contain heap addresses and which one of these is being deallocated at a certain invocation of a deallocation function.

3.4 Candidate chain generation

We define a chain as a sequence of basic blocks that has to be executed in the given order to realize an exploit. In the case of a double free exploit in the CLA setup, we need a chain of basic blocks starting at the entry of an unprotected function that allocates memory, followed by the actual basic block that makes a call to the allocating function and ending with the entry block of the protected function followed by the actual block that calls the deallocation function. The examples given above in fig. 3-2a and fig. 3-2b are taken from the binary file *firefox* in the Firefox browser program. In the case of fig. 3-2a, the first basic block located at 44cd20 is the entry block of the function that allocates the heap memory which our double free exploit

targets. Following this block, we need to execute the basic block that does the actual allocation. That is the basic block at 44cf9b. Following that, we need to execute the basic block that calls the function sub_44d050 which is the basic block at 44ce63. The function at 44ce63 is a protected function that contains an instruction deallocating memory. Following the call to this protected function at the end of basic block 44ce63, after which we execute the entry block of function sub_44d050, we need to execute the block with the call to the deallocation function, which is free in this case, located at address 44cf9f. For a successful double free exploit, we need to execute these blocks in this order. The example in fig. 3-2b is similar to the example in fig. 3-2a except for the fact that it now spans three functions. In fact, the chain can span any number of functions as long as it starts with an unprotected function that allocates memory and ends with a protected function that deallocates memory.

Our chain generation technique uses static analysis. We start by identifying functions that are protected and those that are unprotected. We then proceed to identify function calls in the program that are from unprotected to protected function. Once we have those calls, which we call transfer points, we conduct a breath first search exploration of the call tree on both sides of the transfer points to find the unprotected functions that allocate heap memory and protected functions that free heap memory. If we manage to get to these points, we consolidate and generate all the possible paths from the allocations through the transfer points to the deallocations.

Our chain generation function first starts with a preset parameter for a max length to explore to. After the first chain generation with that parameter, it doubles the parameter and tries to generate chains again. If the number of chains found changes, which means there are more paths to be explored, we double the parameter and try again. This goes on until the number of paths being discovered plateaus which tells us there are no more paths remaining. This tunable parameter determines how fast this phase would be able to generate all the possible paths. Before we arrived at this decision, we designed an exploration method that returned a less than optimal number of paths to reduce the run-time of this phase. We would stop exploration prematurely after a certain number of chains are generated. After performing the

next two phases on these candidate chains, if there is no viable exploit chain found, we would come back to this phase and generate more. However, we realized the runtime of our automatic exploit generation is severely dominated by the second phase, that is the symbolic execution phase. Thus we decided to get all the paths that are available in this stage.

Earlier we defined a chain as a sequence of blocks from an allocating unprotected function to a protected deallocating function. For a complete exploit however, we would need to find a chain of blocks that starts at the program entry point. ACLEG can be setup so that candidate chain generation will also look for a path from a user determined entry point through the the unprotected allocating functions to the protected deallocating functions. Alternatively, the programmer can set the parameters so that each allocation to deallocation chain will reach as far back from the allocation function as can be traced. For instance, even though there is no path from the main function to the allocating function, function A, in the figure above, there is a path from function B to function A. Thus the chain can be expanded. This can help in more precise symbolic execution, which is the next phase of exploit generation. Arguments to a function are taken as symbolic if execution starts from the function. Starting symbolic execution a bit earlier can help turn some of the arguments concrete. Both styles of extending the allocation to deallocation chain on the side of the allocation function are done in the exact same manner as the allocation to deallocation chain generation is done - completely statically.

3.5 Symbolic execution strategies

This chapter would discuss how symbolic execution is used for validating the chains that can be satisfied among the candidates - prioritizing, caching, pre-constraining, multi-threading are some of the techniques that will be discussed.

The chains generated above are not flow sensitive. There is no guarantee that the chain in fig x can actually be executed in the order required. This is because of loops, function arguments and branches that are dependent on variables or con-

stants. For the case of constants, some of the paths might have been removed by advanced compiler analysis but there is no guarantee that is the case with the binary we are analyzing. On the other hand, variables can not be determined ahead of time. Therefore, symbolic execution is an ideal method to add flow-sensitivity to our chains and know what the required inputs are to affect execution of the chain in the order specified.

However, symbolic execution suffers from the path explosion problem and thus can not be relied upon as is. We develop a symbolic execution strategy that combines pre-existing techniques as well as tailor made once to be a tractable solution for verifying flow in a sequence of chains and finding the inputs to satisfy the constraints.

The first attribute of our symbolic execution is the fact that it is pre-constrained by the previous phase of the exploit generation pipeline. We start by getting a state built from the first function in a chain. Then we run symbolic execution until we reach the next basic block in the pipeline. The symbolic execution engine stops if it reaches a block that satisfies the find condition. At that point, multiple states might have reached the find state. Multiple states might have been deadened, i.e. reached a block with no children. Multiple states might have become unconstrained, i.e. the instruction pointer has itself become symbolic. Multiple states might be running an active thread. Regardless, we follow a depth first strategy of stopping every other thread but the once that satisfy the found state. We simply stop the engine, change the find state to be the next block in the chain and continue. The other states, specially the active once which might be relevant later are stashed. If at some point all the states being followed become deadened, we backtrack and reactive a stashed state to explore other paths.

Caching is another method we leverage for optimizing our symbolic exploration. It happens to be the case that multiple different chains sometimes start with a continuous chunk of similar blocks. To optimize those case, we cache states reached with the basic blocks discovered so far used as a key and check with the cache before launching a novel symbolic execution thread.

Through the experiments we conducted, we figured that symbolic execution gets

much more complex if every function call is followed. Our solution is to constrain symbolic execution only to the basic blocks that are part of the chain. To this end, we leverage Angr's hooking mechanism which can replace function calls with simple abstractions. If any of the basic blocks consequently make use of the return register, then that is marked as symbolic. This greatly reduced the run-time of the analysis at the cost of more symbolic variables. More symbolic variables means more likelihood of an unconstrained state, which happens when a target of an indirect jump or call is symbolic. In our analysis, we note what variables are symbolic because of an abstracted function call. If an indirect jump is dependent one of these returned symbolic variables, we will go back and symbolically execute the abstracted function.

Another optimization for the symbolic execution is parallelization. The candidate chains generated in the previous stage can be explored independently in parallel. Symbolic execution on each chain itself can be parallelized. After exploring Angr's inbuilt option to parallelize each symbolic execution engine, we decided to go with parallelizing over the candidate chains as the number of cores available rarely exceeds the order of the number of chains we work with. Angr's inbuilt parallelization tool is exceedingly unreliable and slow.

3.6 Recursive Reaching Definition Analysis

In many instances, it is the case that there are more number of variables in a program than the number of registers available for holding them. Since most computer operations occur on data held in register, the compiler has to decide what variables get to be in the register at different points in a program. Compilers generate instructions for copying data from memory to registers when needed and spilling data from registers back to memory when the data is no longer immediately needed and the register is needed for holding another data. For our double free exploit, we need to fetch what memory holds the critical data, that is the pointers to the heap allocate memory, are stored at. This information is necessary to learn what arbitrary reads and writes are needed for the double free exploit.

Effectively, given a point in code we would like to learn what memory location holds a value equivalent with certain registers. Following an allocation, we track the operations on the return register - which in x86 is *rax* - until we learn a memory location that holds the same value as it. Just before deallocation, we would like to learn what memory holds the same value as the first argument register - which is *rdi* in x86. Since what gets passed to the deallocation function is the heap allocated memory pointer, overwriting the pointer before it gets copied to the argument register would with the value of the heap address from the unprotected section will result in a double free when later the unprotected section conducts its garbage collection.

Reaching Definition Analysis (RDA) is the ideal analysis for this purpose. Specifically, we run RDA in the basic blocks following the allocation function calls following the unprotected sections and just before the deallocation function call function argument fetches in the protected sections of the program. For instance,

Over the course of our exploit generation, we run multiple rounds of RDA By this phase, we would have identified the chains that can be executed in the order required and the inputs to achieve effect their execution. These chains span from the unprotected allocating to the protected deallocating functions. For a full exploit, we need to know where to write and what to write it with. This phase of the analysis achieves that.

The Reaching Definition Analysis we developed is built on top of Angr's Reaching Definition Analysis. Effectively, it allows to efficiently learn where the result of a specific function is written to in memory and what the arguments of a specific function call come from. The later helps know where the argument for the free comes from. In other words, this is the stack address that contains the heap address to the memory we are freeing. This is the location we should overwrite with the memory address that is the target of our double free. The former is used to know where the heap addresses allocated by unprotected functions are kept on the stack. This memory contains the target of our double free.

In the figure above, we have an excerpt of a code where in the last basic block there is an access made to a buffer and we would like to know its address with respect

to rsp. But the buffer access instruction itself does not say much. The base of the memory access is rx. In basic block 1, we see the instruction $leary, 0x10[rsp]$. In basic block 2, we see $movrx, ry$. If we put an observation point just before the end of the last basic block and did RDA, we would see a definition of rx pointing to basic block 2. But that tells us nothing about what rx contains. We would need to do another RDA with an observation point just before the instruction in basic block 2 to learn what ry and thus ry contain. This is what we call Recursive Reaching Definition Analysis.

The base case for stopping the recursion is when the definition observed is what is called an ultimate source. This is either a location that is outside the context of analysis or a memory location. Which base case is used depends on a user provided parameter. The design decision to have this is motivated by practical demands of exploit generation. For double free exploit generation, we found it important to learn where on the stack the argument for the free function comes from. The memory location being freed is either stored in the stack or is provided as an argument to the function. In the former case, stopping recursion when the source is a location on the stack is the right strategy. In the latter one, the recursion should stop when the source is outside the context. Moreover, our RDA analysis is limited to only the blocks in a chain. Given a state reached after executing all the blocks in a chain, we extract the exact sequence of blocks that were executed and from those blocks we create a linear graph. We apply Angr's RDA over that graph instead of a full function or program graph which would have been too inefficient.

By the end of this phase, we have learned what blocks to execute from the first phase, what inputs to provide to effect their execution from the second phase and what arbitrary read and writes to conduct by the third phase. We have everything we need for a full exploit.

Chapter 4

Implementation

ACLEG is built on top of Angr. Angr's architecture independence, open-source nature and the several analyses methods it has made it a compelling choice. The CFG we rely on for our analysis is generated using Angr's *CFGFast* CFG generation. For the first phase of ACLEG , we generate a call relationship map and navigate it along with the CFG to extract all the candidate chains available. For the second phase, we take these candidate chains and apply our symbolic execution algorithm on them. Our symbolic execution algorithm is built on top of Angr's symbolic execution engine. For the third phase, we take the satisfiable chains from phase 2 and apply RDA on them to learn critical information for a double free exploit. Our RDA analysis is also built on top of Angr's inbuilt RDA analysis.

4.1 General Flow

As shown in 4-1 below, execution starts by invoking the *main.py* script. The script should be invoked with the name of the main binary file, the names of shared libraries to be loaded with it for analysis, the log file to be used for outputting log entries and a json file that will contain the results of analyzing the binaries. *main.py* invokes the *cla_json* script which is responsible for generating the json file containing the output of the CLA analysis. *cla_json* does this by creating an *AutomaticCLA* object, getting the results of the CLA analysis and dumping the results to the specified json file. Once

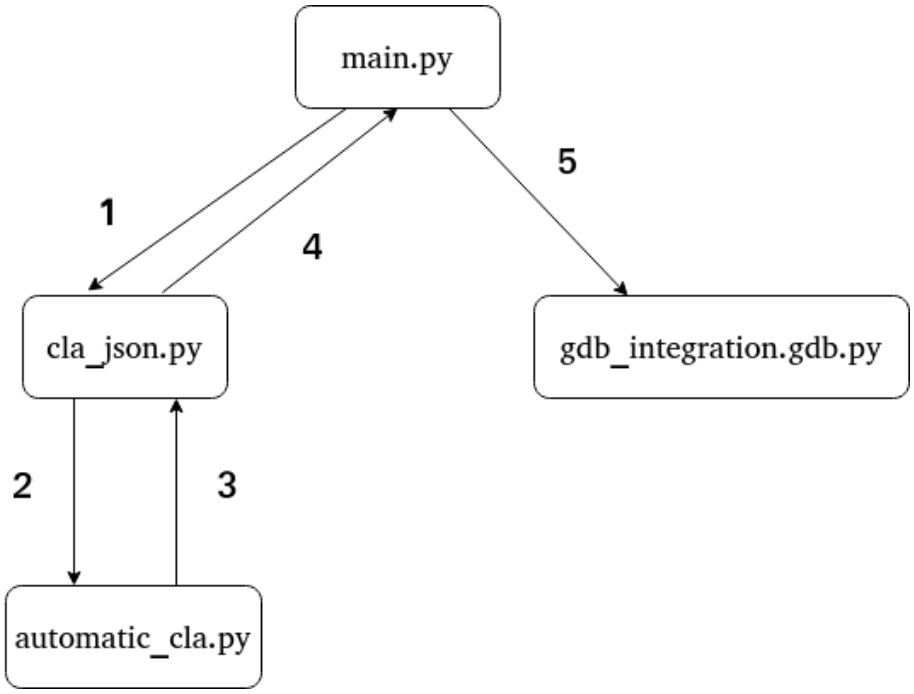


Figure 4-1: **step 1**: *main.py* invokes *cla_json.py* with the right arguments; **step 2**: *cla_json.py* creates an *AutomaticCLA* class and invokes the *find_double_free_vulns* method; **step 3**: *find_double_free_vulns* returns a dictionary with a list of the vulnerabilities and the necessary conditions for exploiting them; **step 4**: *cla_json.py* dumps these into a json file and returns the name of the file to *main.py*; **step 5**: *main.py* invokes the gdb automation script with the name of the json file as an argument, *gdb_integration.gdb.py* automates the exploitation of the vulnerabilities

the *cla_json* script is done running, *main.py* invoked the *gdb_automation.gdb.py* script which reads the CLA analysis result from the json file and executes the attacks using the information therein.

CLAChain is a class for representing one single chain of blocks. It abstracts away the important details of managing a chain while it is being worked on through the four phases of the analysis.

AutomaticCLA is the major class of the tool. AutomaticCLA is instantiated with the name of the main binary and its dependencies. Other parameters like log level and log file are also specified.

CFG generation: Angr provides two options for generating CFG's - the emulated and the fast CFG methods. Emulated CFG is built by symbolically executing the entirety of the program taking a trace while fast CFG is built by statically analyzing the code. Emulated CFG produces a much more detailed CFG and it is required for Angr's DDG analysis. However it is extremely slow and unreliable. Run on a 32-core, 64GB memory computer, it consistently fails to complete building even for simple binaries. Thus we opted to rely on fast CFG for the analyses in this work. If CFG emulated ever becomes required and tractable, AutomaticCLA allows the user to specify the CFG type and constructs that. It is set to CFG fast by default.

Call relationship maps: For the candidate chain generation phase of exploit generation, it is important to have a graph of function call relationships. It is often essential to know what functions call a specific function and what functions are called by the function. Additionally, we also need the basic block addresses of the call sites associated with each call we explore. Thus, AutomaticCLA keeps two dictionaries. One, called *callee_to_caller*, keeps a mapping between a function and its callers. The second, called *caller_to_callee* keeps a mapping between a function and its callees. These dictionaries are constructed in a completely static manner by walking the CFG tree of the project.

Function classification: The next step in setting up the analyses is determines what functions are protected, i.e. have standard mitigation techniques for unsafe languages are deployed, and what functions are not. We accomplish this using a simple method of scanning the prologue of functions and matching them to fingerprints of known mitigation techniques. For instance, use of stack canaries can be detected by looking for a copy from a memory address in the fs section to a memory address on the stack in the prologue of a function. Use of control flow integrity can also be done in a similar manner. This method managed to completely profile all the functions found in the commercial software programs we experimented our tool on.

Besides profiling individual functions as unprotected and protected, we also analyze the functions calls made by each function and identify the points of transfer. These points are points where an unprotected functions make calls to protected functions and vice versa. These points of transfer are identified and stored in a dictionary for easy access in the later stages of the exploit generation.

Mallocs and frees: For executing a double free attack described above, it is important to find an allocation made by an unprotected language connected via an execution path to a deallocation made by a protected function. Thus the next stage of our exploit construction process is to find the unprotected function allocations and the protected function deallocations. This is done by looping through every function and analyzing the function calls made by each function. It is assumed that the allocation and deallocation functions are known. Usually, these are *malloc* and *free*. Often, many programs deploy wrappers around these.

4.1.1 Candidate chain generation

Once we identify the unprotected function allocations and the protected function deallocations, the next stage is to find a chain of basic blocks that connect the two. We do this by starting from points of transfer. Then we generate two classes of chains: chains for unprotected functions that allocate memory to the point of transfer and chains from the point of transfer to protected functions that deallocate memory. These

chains are generated by performing a binary first search on the function call graph where the longest distance covered is a user specified parameter. All the combinations of chains from the two chains are legitimate candidate blocks for an exploit.

Once all of these chains are generated, we create *CLA_Chain* objects for each one of them. The *find_double_free_vulns* function loops through these chains and calls *find_path* on each one of them to add flow-sensitivity.

4.1.2 Flow-sensitivity via symbolic execution

Note that the chains above are generated by a purely static exploration of the call graphs. There is no guarantee there is a way to make the chain execute, i.e. execute the chain in the order specified. In other words, the chain is flow-insensitive. The next phase is to add flow sensitivity. This is achieved by using symbolic execution which is performed in the *find_path* function in the *AutomaticCLA* class.

Our implementation of symbolic execution builds on Angr's symbolic execution engine. *find_path* takes a *CLA_Chain* object along with parameters that determine what optimizations get applied. It starts by getting a new *ProgramState* object from Angr for the first function of the chain by using the *call_state* method which initializes a state from a class. Then a *SimulationManager* object is created to manage the symbolic executions of the state we just initialized and its spawns in the process of symbolic execution. The *SimulationManager* class contains the *explore* method that takes *find* and *avoid* conditions. *explore* executes the states being manages until they end or the find/avoid conditions are met. In either case, exploration is stopped, states are classified accordingly and returned to the user. ACLEG calls this method with the *find* condition being whether a state reaches the basic block right next in the chain we are trying to add flow-sensitivity to.

Path explosion is the major reason symbolic execution does not scale well. In our implementation, we took a few steps to mitigate the issue and make symbolic execution tractable. Our first observation was that following every function call was not necessary. Every function call would roughly double the number of paths to be explored. Thus we made use of Angr's hooking mechanism to abstract away function

calls to functions that are not part of the chain before we call Angr's explore method. If the rest of the code relies on a value returned from these function calls, the values returned are taken as symbolic and can later be solved to satisfy constraints.

As mentioned in the section above, $find\_path$ is called by the $find\_double\_free\_vulns$ method once for each chain. Each call could take quite a considerable amount of time. On the other hand, most modern computers are equipped with several physical computing cores and the problem at hand is parallelizable. Thus, $find\_double\_free\_vulns$ allows the user to pass a parameter specifying if parallel exploration of the chains is desired. If so, one core is used for executing one invocation of $find\_path$. For n chains, this achieves a $O(T * (logn))$ span and $O(T * n)$ work where T is taken to be an average time a single path of symbolic execution takes. This is a work-efficient parallelization.

4.2 *RDA_Explorer*

The *RDA_Explorer* class builds on the inbuilt reaching definition analysis framework of Angr to implement a recursive reaching definition analysis that can be used to capture the arguments to a function and the stack spaces where returned values are written to. In the context of double free exploit generation, it is used to detect where to conduct arbitrary reads and writes. In vector bounds bypass attacks, it is used to understand what parts of the code try to access a vector and where to conduct the arbitrary write. In this section, we discuss the implementation details of the class.

4.2.1 Recursive RDA

A novel contribution of our work in this thesis is recursive RDA that is restricted to a set of blocks as outlined in the design chapter above. Its implementation in the *RDA_Explorer* class is the function *ultimate_def_val*. *ultimate_def_val* takes in a definition the ultimate source of which we are trying to find as well as a definition of what an ultimate source is. By default, it is taken to be a source from outside the set of blocks we are restricting our RDA to, which in Angr is specified by the

ExternalCodeLocation class. Alternatively, it can be made to track the chain of definitions until a value is copied from memory to register. This design is based on the practical demands of our exploit generation technique that necessitated RDA to begin with as will be discussed below. Given these parameters, *ultimate_def_val* starts by getting the instruction where the given definition happens at. After investigating the instruction, if it is an ultimate source as defined by the other parameter, it stops and returns. Otherwise, we recursively invoke *ultimate_def_val* to learn the ultimate source of the source operand in this instruction is, which by extension is the desired ultimate source. To do this, an observation point is set just before the current instruction and Angr's RDA engine invoked to find the definition of the source operand. Once we get this definition, we simply call *ultimate_def_val* with it as an argument and passing on the other parameters as they were.

4.2.2 Arguments to a function

Finding arguments to certain functions is an important part of our exploit generation mechanism as discussed in the design section. Our implementation of our solution is in the *find_args_to_func* method in *RDA_Explorer*.

find_args_to_func takes the address of the basic block that makes the function call we are analyzing, which is assumed to be one of the blocks in the block chain that form the space of our RDA analysis. Note that the call instruction must be the last instruction in the basic block. Then an observation point is set just before the call instruction and Angr's RDA analysis invoked. Following this, we invoke a call site analysis method called *analyze_callsite*. Based on the calling convention for the architecture, this method goes through the definitions we found from the RDA and if the definitions are to an argument location - either register or stack memory in most architectures - then it gets registered as an argument. The last step is to go through all of the arguments we found and make sure they have ultimate source definitions. To this end, *ultimate_def_val* is invoked on all the argument definitions and the results returned.

4.2.3 Returned values

Many compilers compile binaries where the return value of a function call, often in a register, is copied to memory to free up registers which are often competed for by variables. In this section, we discuss how we tackled tracking where exactly this happens on memory. We are especially interested in learning what address on the stack ends up containing a return value of a specific function call and what its offset with respect to rsp or rbp is.

In our tool, this is implemented in the *find_where_returned_val_is_written* method of *RDA_Explorer*. The method takes in the address of the basic block where the function call is made. The basic block right next to the function call block is guaranteed to be a single basic block and the block to be executed right after. This is where we start our analysis. If there is a simple instruction where the architectural return register is simply copied to a memory on the stack (given as an offset from rbp or rsp), the method can simply learn that and return. If there is obfuscation of the memory location it is being copied to by using a memory base stored in another register, then we invoke the *ultimate_def_val* method to learn the ultimate source of the base register.

4.2.4 Performance optimization

Over the course of analyzing a binary, multiple *RDA_Explorer* classes are instantiated and multiple Angr inbuilt RDA calls are made. Many of these RDA calls are however repetitive. A major performance optimization we made is to cache all the RDA states we generate. It is keyed with the a hash of $(observation\ point, blocks\ in\ RDA)$. The cache is contained in the main $AutomaticCLA$ object and is accessed through the *get_rda_state* method. When an RDA state is required, this method is invoked with the the necessary arguments (observation point and the blocks in the RDA). The cache is first checked before resorting to calling the RDA analysis of Angr.

4.3 Putting everything together

To recap the major phases of exploit generation, we first begun with generating static, flow-insensitive chains. We proceeded to find the chains that are satisfiable by performing symbolic execution. For double free exploits, we then proceed to use *RDA_Explorer* for finding the arguments for the deallocation call and the stack location that contains the allocated memory by finding what memory address receives the result of the allocation function.The next step is to use gdb to automate actual exploits using the information we have collected.

We achieve this using a simple python script. We first define a JSON interface between the exploit generation mechanism and exploit execution in gdb. An example is given in fig x above. The *read* section specifies what memory is to be read. The *write* section specifies what location is to be overwritten. The *breakpoints* section specifies where to set breaks. The *binary* section specifies what binary is to be loaded. The *deps* section specifies what other shared libraries are to be loaded with the main *binary*.

The JSON is generated by the *double_free_vuln_json_dump* function located in *cla_json.py*. This function takes in the binary, its dependencies and the json file name to dump to. After creating a binary and generating the vulnerability summary, it goes through each of them and create the required JSON for each. These are then dumped to the file.

gdb_integration.py takes care of the final step. Given a JSON file name containing the vulnerability specification, it first loads the JSON. After this, it uses a python operable feature of gdb to generate and execute the exact sequence of gdb instructions to peform the desired attacks.

4.4 *Binary_Groups*

Almost all commercial software products come with a variety of binaries with inter-dependence between them. In Angr, it is possible to dictate if a binary is to be loaded

along with all, partial or none its dependencies. Since binary analysis is a computationally demanding task, sometimes it might make more sense to load a binary alone and analyze it. Sometimes that can be too constraining and it might be desired to load a partial or complete list of dependencies for analysis. The *Binary_ Groups* class in our tool makes it easy to manage a group of binaries and apply the desired analyses in a tractable manner.

A typical use case for this *Binary_ Groups* class is when we want to slowly add more dependencies to an analysis until we reach a desired result. The group takes care of creating all different combinations of binaries to be loaded together and loads them progressively from the smaller to bigger sets until a requirement we desired is achieved. Additionally, it can help generate several statistics on a group of binaries that make up a program. We made use of this group for the statistics we generated for chapter 5.

Chapter 5

Evaluation

We tested ACLEG both on microbenchmarks we wrote and commercially available software programs. For the former, we made a simple program partially written in C and partially written in Rust. It contains paths that can be executed for double free exploits. We are using the microbenchmarks to evaluate our tool for accuracy. For the latter, we used two famous programs that are available commercially: Firefox and Fuchsia. Our tests were performed on a 32-core, 2GHZ, 64GB RAM x86_64 hardware running an Ubuntu Linux 5.15.0-40-generic kernel. We use these two complex programs to evaluate the scalability of our tool and the potential for CLA.

5.1 Full-exploit on a sample code

The sample code we constructed to test our code is a simple combination of Rust and C. There are two allocations (one for a vector and one for in a Rust function and a deallocation in the C function. The first allocation is for a struct object and the second allocation is for a vector. The allocating Rust function calls the deallocating C side. The C side is compiled with stack canaries and CFI enabled while the Rust side is enabled without these protections. The goal is to escalate a memory corruption on the C side to a double free on the Rust side. ACLEG finds both paths for exploit in this case in less than 60 seconds and the end-to-end exploit simulation in gdb works as expected, generating a double free in both cases.

binary	protected funcs	unprotected funcs	protected to unprotected calls	unprotected mallocs	protected frees
firefox	423	2487	282	141	473
libmozavutil.so	140	946	129	2	1
libmozgtk.so	0	17	0	0	0
libnss3.so	505	3190	247	0	0
liblgpllibs.so	4	280	0	0	0
libfreeblpriv3.so	226	1318	94	5	38
libplds4.so	2	93	0	0	0
libsoftokn3.so	176	1390	77	1	4
libmozsqlite3.so	405	4234	762	1	0
libnspr4.so	113	1322	157	13	36
libmozsandbox.so	122	644	24	0	0
libnssckbi.so	72	693	31	0	0
libmozwayland.so	0	113	0	0	0
libplc4.so	0	133	0	2	0

Table 5.1: *Statistics on the functions contained in different binaries in the Firefox program*

binary	max length of a path	chains generated	time for generating	satisfiable paths	time for symbolic execution
firefox	3	15	0.401	0	1853.015
libnspr4.so	6	339	0.25	0	38209.74

Table 5.2: *Analysis results on binaries in the Firefox program*

5.2 Results on Firefox

The Firefox code base is composed of more than 18 different binary files. Among these, *libxul.so* is the biggest binary and contains the most functions in the program. However, Angr was unable to load libxul.so and we were unable to conduct our experiments on it.

5.1 demonstrates the CLA capacity of the binaries we analyzed. The firefox and *libnspr4.so* hold the most potential for CLA exploitation. The firefox binary contains more than 473 deallocations from a protected function, 141 allocations from unprotected functions and 282 cross-language function calls. The corresponding figures for

libnspr4.so are 36, 13 and 157.

5.2 shows the data collected through the three phases of our exploit generation technique. The max length of a path is the maximum number of function calls on the paths discovered and run times are given in seconds. Only two binaries contained candidate chains. For firefox, our tool found the optimal chain length to be 3 function calls long. All the 15 candidate chains were recovered in less than 0.401 seconds. The symbolic execution phase was able to complete running in 1853 seconds, for an average of 123.5 seconds per candidate chain. However, we were unable to obtain an exploitable chain. We address why this might be and recommend how it can be improved in chapter 6. The data is similar for libnspr4.so, where we found 379 candidate chains but were still unable to generate a satisfiable path.

5.3 Results on Fuchsia

Table 5.3: Statistics on the functions contained in binaries in the Fuchsia program

binary	protected funcs	unprotected funcs	protected to unprotected calls	unprotected mallocs	protected frees
usb-cdc-function.so	85	493	30	3	5
qmi-transport.so	166	630	69	3	5
dwc2.so	210	530	31	3	8
bt-hci-atheros.so	192	858	82	3	25
usb-peripheral.so	179	795	82	3	5
zxcrypt.so	461	1801	180	2	20
ddk-not-fallback-test.so	157	655	56	2	20
virtio_scsi.so	120	550	35	2	1
rndis-host.so	54	342	18	3	5
brcmfmac-test.so	744	2657	404	4	9
ums-function.so	38	180	42	1	5
iwlwifi.so	766	2034	350	4	730
usb-mass-storage.so	74	491	20	3	5
fvm.so	508	1908	235	2	20
display.so	1433	4160	602	3	20
aml-i2c.so	70	336	25	2	2
test-driver.so	206	848	77	2	20
usb-bus.so	213	921	116	3	5
pc-ps2.so	589	1851	201	2	20

Continued on next page

Table 5.3: Statistics on the functions contained in binaries in the Fuchsia program (Continued)

i2c-hid.so	282	1239	128	2	3
ddk-environment-test.so	222	913	76	2	20
ftdi-fake-usb.so	42	319	18	3	5
sherlock-camera-controller.so	1159	3871	418	3	20
ddk-fallback-test.so	157	655	56	2	20
VkLayer_compact_image.so	458	1502	179	3	20
fake-usb-cdc-acm.so	42	328	20	3	5
i2c.so	999	2972	503	2	1
structured_config_receiver.so	1296	3715	595	2	20
block.core.so	335	1355	203	2	20
bt-transport-usb.so	108	580	36	3	5
ftdi.so	189	845	62	3	5
virtual-bus-tester-function.so	34	270	20	3	5
usb-audio.so	680	2074	284	3	5
gdc.so	545	2216	208	3	20
usb-cdc-ecm.so	18	115	2	1	5
usb-cdc-acm.so	62	434	26	3	5
xhci.so	1036	5107	423	3	7
libmsd_arm_test.so	2148	5172	1008	2	21
services_root.so	454	1609	127	2	20
ath10k.so	213	637	117	3	25
virtual-bus-tester.so	103	614	33	3	5
goldfish_sensor.so	1028	3279	368	3	20

Continued on next page

Table 5.3: Statistics on the functions contained in binaries in the Fuchsia program (Continued)

bt-hci-emulator.so	1820	6011	1010	3	21
interop_root.so	1560	5022	756	2	20
libmsd_intel_test.so	1898	5192	1070	2	21
gpt.so	196	898	73	2	20
asix-88772b.so	18	101	4	1	5
usb-two-endpoint-hid-fake-usb.so	45	323	21	3	5
nvme-cpp.so	425	2070	143	2	1
fake-asix-88179.so	95	519	37	3	5
usb-peripheral-test.so	45	360	22	3	5
compat.so	2361	9247	1235	2	20
VkLayer_image_pipe_swapchain.so	701	2301	279	3	20
sysmem.so	2052	5880	1001	3	20
usb-hub.so	296	1587	108	3	5
bt-hci-intel.so	288	1172	106	2	20
asix-88179.so	104	554	45	3	5
mbr.so	166	755	60	2	20
VkLayer_khronos_validation.so	8666	14117	6961	3	35
goldfish-display.so	508	2023	224	2	20
usb-hid.so	49	375	17	3	5
nxp.so	139	511	83	2	2
ddk-fallback-test-driver-module.so	160	664	55	2	20
libmsd_vsi_test.so	1975	5031	1046	2	21
ge2d.so	1038	3222	306	3	20

Continued on next page

Table 5.3: Statistics on the functions contained in binaries in the Fuchsia program (Continued)

ddk-firmware-test.so	240	947	90	2	20
brcmfmac.so	735	2640	402	4	9
composite_root.so	1360	4226	619	2	20
rndis-function.so	114	648	52	3	5
dwc3.so	177	601	55	3	7
bt-host.so	4841	15444	2629	3	20
imx227.so	499	1955	189	2	20
goldfish_control.so	1000	3156	374	3	20
usb-hci-test-driver.so	140	729	43	3	5
usb-virtual-bus.so	121	672	57	3	5
wlan.so	136	48861	64	5	1

Fuchsia is a multi-language, open source OS written in C and Rust. It is composed of more than 300 binary files. We run our experiments on these binary files and report all relevant information in 5.3. In Fuchsia, our candidate generation algorithm failed to generate any number of candidate chains. This is caused by the fact that the binary files in Fuchsia interact through the PLT table which is beyond the scope of our work.

5.4 Evaluation of chain generation

We discussed the design and implementation of the candidate chain generation phase earlier. In this section, we discuss the performance and effectiveness of the different optimization we introduced.

The first optimization we introduced was to allow exploring paths of more than one function jump in the path. Constraining ourselves to single function jumps, which means we are only interested in an unprotected function that allocates memory calling

a protected function deallocating memory, we found 7 such paths in the firefox binary and 2 in the toy example we explored. Every other binary contained 0 candidate chains. After introducing this optimization, we found 15 chains in the firefox binary and 379 chains in the *libnspr4.so* binary.

Following this up, our design allowed constructing chains strictly within the unprotected section from the allocation function to the transfer point and strictly within the protected section from the transfer point to the deallocation function. Applying this method, we discovered 7 chains in the firefox binary. Relaxing this constraint to allow jumping between the protected and the unprotected sections on the path from the unprotected allocation function to the transfer point and the transfer point to the protected deallocation function increased the number of paths by more than a factor of 2. The run time in the latter case is reported in the table 5.2 as $0.401 secs$ while it took $0.24 secs$ for the former case. This is not a significant performance cost as the run time ends up being dominated by the symbolic execution phase.

Finally, we introduced a binary search approach to determine the maximum length to explore to in candidate chain construction. Prior to this optimization, the generator function was a python generator object that generated more paths as required. This design was motivated by the assumption that the symbolic execution phase would request more paths as desired and slowly step up the length as needed. However, this did not have the desired performance benefits as the symbolic execution phase completely dominated the run time. As shown in 5.2, the system spends nearly a 100% of the time in phase 2.

5.5 Performance of Symbolic Execution

Our symbolic execution algorithm is designed for a specific case where the exact sequence of blocks to be executed are known. Finding this sequence ahead greatly improves the performance of the symbolic execution phase. To measure how much performance this chain constraining step provides, we attempted to run a semi-constrained symbolic execution algorithm where only the beginning block and the

Strategy	cc 1	cc 2	cc 3	cc 4	cc 5	cc 6	cc 7	cc 8	cc 9	cc 10	cc 11	cc 12	cc 13	cc 14	cc 15
Full algorithm	14.40	3.95	4.05	>600	0.05	0.04	2.23	2.23	3.67	3.94	2.06	4.01	4.16	3.96	3.81
Without caching	14.90	3.78	3.7	>600	>600	>600	2.25	1.95	3.62	3.72	2.27	4.21	4.00	4.07	4.07

Table 5.4: *Performance of the full algorithm compared to that without caching*

end block are given. Even disregarding the fact that among the found paths, one would later need to filter the ones that contain the allocation basic block in the path, this turned out to be impractical as we expected. Every Firefox and Fuchsia binary we run it on run out of the 10 minute limit and the job had to be killed. For the toy binary, the unconstrained state took twice the time needed for the constrained case.

Another aspect of our symbolic execution algorithm is the backtracking based DFS exploration. Without this aspect, whenever there is an unsatisifiable path the algorithm would have to start all over again. To precisely determine how much this step helps, we run a modified version of our symbolic execution algorithm does not implement a backtracking approach but goes back to the beginning to start exploration again when a link in the chain become unsatisfiable. Since an infinite loop is possible if we allow already explored paths to be explored again, we note the states we have seen already and do not follow them when exploring again. The results again showed a clear advantage to backtracking as would be expected. This algorithm failed to complete symbolically executing a chain in the 10 minutes limit we put. Only one chain times out in the case with backtracking enabled.

Caching is another performance feature of our algorithm. To measure its benefits to performance, we run our symbolic execution algorithm with this featured disabled on all the 15 candidate chains in the firefox binary. The results are given in table 5.4. Candidate chains 4, 5 and 6 share the first three blocks of the chain. Since the functions these basic blocks are contained are complex, our symbolic execution does not finish running in 10 minutes. The caching optimization can recognize this and avoids running on 5 and 6 again. Many of the other candidate chains demonstrate some performance benefits as well. In some instances, the basic blocks at the begin-

ning of the chain are not in the cache and thus introduction of caching does not only not improve performance, but causes a minor loss of it.

Parallelization is the last performance optimization we introduced to our symbolic execution algorithm. While it is not perfectly functional as is, we suspect its unreliability as of now is caused by race conditions within Angr which we think could be addressed in the future. Thus, we think it is worth evaluating its performance. We run a symbolic execution with one thread for each one of the 15 candidate chains found in the firefox binary. The run time was 600.4 seconds. This indicates at least one of them exceeds the time limit and had to be killed. Running symbolic execution sequentially over the 15 candidate chains took 1852.75 seconds. This is a significant performance gain. Parallelization reduced the performance by more than 67%.

Chapter 6

Discussion

Our work had several limitations. Primarily, we failed to produce a full working exploit in a commercial software. Due to the scalability and reliability limitations in Angr, we were unable to conduct as complete an analysis on the programs we analyzed. Our analysis does not take advantage of advanced analysis techniques like Data Flow Analysis. Our chain construction does not take advantage of function calls that happen through the PLT table to functions contained in other binaries. We targeted only vector and double free exploits among the five discovered in prior work. Additionally, we had to make strong assumptions to conduct the double exploits in the sample toy binaries we analyzed. In this chapter, we discuss in detail the limitations of our work, what caused them and how future work can help address them for a more complete Cross-Language Analysis. We also discuss the 10 minute limit on invocations of the symbolic execution.

6.1 Limitations

One of the limitations in our work is the incompleteness of our analysis. As shown in 6.1, a significant portion of the function calls in Firefox is cross-binary. In average, more than 37% of the function calls are from a function in one binary to one in another. As different binaries tend to contain code coming mostly from one language, this is a sever limitation in a Cross-Language exploit building tool. We found no way

binary	total calls	plt calls	plt calls percentage
firefox	8045	2529	31.44
libnssckbi.so	1076	119	11.06
libsmime3.so	2265	1675	73.95
libplc4.so	48	42	87.5
libnss3.so	11445	6035	52.73
libmozgtk.so	3	1	33.33
libplds4.so	56	19	33.93
libsoftokn3.so	4528	2128	47.0
libnssutil3.so	1445	598	41.38
libssl3.so	5942	3213	54.07
libmozavutil.so	1876	633	33.74
liblgpllibs.so	216	97	44.91
libmozsandbox.so	2280	881	38.64
libipcclientcerts.so	2572	128	4.98
libnspr4.so	3049	1189	39.0
libmozavcodec.so	13243	3047	23.01
libmozwayland.so	3	1	33.33
libfreeblpriv3.so	5399	760	14.08
libmozsqlite3.so	16447	1397	8.49

Table 6.1: *Statistics on cross binary function calls*

of navigating through the PLT table to take advantage of these function calls in Angr. Additionally, we were unable to take advantage of advanced analysis techniques like Data Flow Analysis and Value Flow Analysis that Angr provides. However, these analyses can only be applied if the CFG is CFGEmulated. As noted above, Angr builds CFGEmulated by running a complete symbolic execution of the program. This is infeasible even for the small toy binaries we conducted our experiments on. Angr was also unable to load big binaries. In Firefox, the libxul.so binary file is more than 15 times bigger than the other binary files combined. Angr was completely unable to load this binary. These issues, we believe, severely limited the scope of our work and the results we obtained.

Another limitation of our work was the strong assumptions we made about the attacker. We assumed that the attacker can arbitrarily read or write any memory location from the memory unsafe sections of the binary. This is an unrealistic assumption. Usually, attackers have a few memory bugs that allow them to write to or read from a limited set of locations. Buffer overflows, heap overflows and memory leaks are common memory corruption techniques that make this possible. As many tools exist that allow automatically discovering these memory corruptions, our work did not focus on them. But integrating them into a CLA framework would be a great future work and will be discussed below.

Another possible limitation of our work is the 10 minute limit we put on our invocations of the symbolic execution engine. We settled on this limit after running symbolic execution on some of the candidate chains. We invoked symbolic executions with unlimited time for the chains in *firefox* that are normally stopped by the 10 minute limit. The exploration was unable to complete even after 12 hours. We inspected the binary code for these paths and we did not find anything peculiar that should cause it. Our suspicion is that the Angr engine is unable to process some instructions. We decided a 10 minute limit because most of the chains we investigated were done with their exploration with much less time.

6.2 Future work

Including concrete memory corruption bugs: In the work we did in this thesis, we assumed the attacker had access to an arbitrary read/write vulnerability in the protected section of the code. In reality, there are usually a few basic blocks where the attacker can execute a memory corruption to read or write. This could for instance be a buffer overflow. Given such corruption bugs located at certain basic blocks in the code, ACLEG can be adapted to include finding paths to these basic blocks. For instance for a double free cross-language exploit, we needed to perform to overwrite the argument to the deallocation function. This can be done just before the basic block containing the deallocation function is executed. Thus the chain can be changed to include a path from the entry of the function containing the deallocation basic block to the entry of the function containing the memory corruption bug to the basic block within that function with the actual memory corruption bug back to the deallocation basic block itself. Now the memory corruption can be used to overwrite the argument for the deallocation call and thus execute a double free exploit.

Loading more binaries than one: As shown in 6.1, more than x percent of cross-language calls happen with the two functions in two different binary files. Since Angr does not provide an easy way to navigate the Procedure Linkage and Global Offset tables, our analysis was constrained to CLA's that are entirely contained within one binary file. This is a serious limitation of our work. We believe a lot more paths can be uncovered and exploited if routing through the PLT and GOT tables is enabled in the analysis. We leave this for future work.

A more stable disassembly tool: Angr as a binary analysis tool, even though among the best available, has serious limitations. In the course of our thesis, we run into several issues arising from its instability. For instance Angr could not load libxul.so, the Firefox binary file containing more than 93.5% percent of the code. Additionally, race conditions arising from within Angr limit the amount of parallelizing we could hope to achieve. Despite our best attempts, we could not avoid these race

conditions due to the complexity of Angr's design. This means there is a lot of space for improvement in the binary analysis tool space and we hope future work addresses that.

Other types of CLA: Our thesis was limited to uncovering double free and vector bound bypass attacks. This is only a limited set of attacks that can be automated. In fact, most of the work we have done can be easily adapted to working on a variety of attacks. Future work can be in the direction of expanding and building on our tool to extend the analysis to all the known CLA attacks.

CLA in multi-core computing: A completely unexplored direction for CLA's is in multi-core computing setups. If one thread is running a piece of code with different assumptions about security from another thread, this can open up a whole class of CLA attacks than possible in a single core setup where only one piece of code can run at one time. We leave this for future exploration as well.

Chapter 7

Related Work

Due to the inability of mitigations to secure fundamentally memory-unsafe languages like C/C++, many programmers have shifted to using languages like Go/Rust which are memory-safe [3, 43]. As a significant portion of the existing software infrastructure is written in C/C++, programmers often leverage the foreign language compatibility features of these memory-safe languages to write software that is written in multiple languages. However, recent work has shown that this is not secure. Cross-Language Attacks [44, 54] are a new class of attacks that leverage the mismatch of threat models in programs written with different programming languages to undermine the security guarantees of both programming languages. CLA attacks can be broadly classified into two: those that use corruptions in the memory-unsafe side to undermine the security (in particular, the memory safety) of the memory-safe side and those that use the safe side (that typically deploys no mitigations) to circumvent the mitigations deployed in the unsafe side.

In order to discuss the first class of CLA attacks, we next discuss two categories of memory safety vulnerabilities commonly found in unsafe code: spatial exploits and temporal exploits. Moreover, we include discussion of memory safety defenses applied to unsafe code. Afterwards, we discuss a number of control-flow hijack attacks and mitigations commonly deployed on the unsafe side that a CLA attacker can circumvent through the safe side. Finally, we end this section with a discussion of automatic exploit generation techniques that an attacker can leverage to automatically create a

CLA attack.

Spatial exploits occur when a memory access exceeds some region where data is held. Typical examples of this are stack and heap buffer overflows [53, 34]. Ever since their introduction, programmers have introduced various mitigation techniques to defend against spatial exploits. Fat-pointers are one of the first recommended solutions [9]. The fat-pointer approach is to keep metadata associated with every pointer that determines the lower and upper memory bounds the pointer is allowed to access. CCured [48] retrofits legacy C code by adding fat-pointers. Cyclone [36] is a dialect of C that automatically implements fat-pointer mitigation. Both of these solutions change the structure of pointers by appending the metadata to each pointer. These solutions require source code annotation and are not binary compatible as the pointer representation has to be changed. SoftBound [47] is a related solution which does not change the pointer representation itself but holds the metadata in a hash-table. The code is instrumented to verify safety by querying the hash-table before every time the pointer is used. Despite its security guarantees, SoftBound is not a practical solution as its performance overhead is more than 67%. HardBound [25] is a hardware solution for bounding pointer accesses with minimal software modifications. Despite its low performance overhead, HardBound still suffers from the compatibility problems of using pointer bounds. An alternative solution to bounding pointers is bounding objects. In objects bounding, the metadata is attached to objects and not pointers. If a pointer is used to access an object, the metadata of the object is checked for bounds. This approach preserves compatibility. The first such solution was a GCC patch by Jones and Kelly(JK) [37]. JK suffered from performance issues, with up to 11-12x overhead, and had difficulty recognizing a pointer going out of bounds by more than one byte without being dereferenced. A more complete objects binding solution was later introduced in CRED [57], which reduced the overhead to up to 2x. A further optimization of JK by Dhurjati et al. [40] reduced the overhead to 1.2x by performing static analysis to reduce the amount of metadata storage needed. Baggy Bounds Checking (BBC) [7] and PAriCheck [68] are other implementations of object bounds checking that restructure the location of objects such that it is easy

to calculate their bounds, trading memory for performance.

Temporal safety is another class of memory corruption errors. Use after free vulnerabilities occur when a pointer is used after being freed [41]. Double free vulnerabilities occur when a pointer is freed twice [17]. Some of the first solutions to enforce temporal safety use special allocators to limit space reuse. Cling [6] reuses space only for objects of the same type, limiting the possible use-after-free vulnerabilities. Object based approaches keep a shadow memory space to keep track of space that has been deallocated, preventing use-after-free bugs. Valgrind's Memcheck [50] and AddressSanitizer [62] follow this strategy. These two solutions both suffer from high performance overheads. An alternative solution is to associate allocation information to individual pointers. This is done in CETS [46].CETS and SoftBound offer protection against both temporal and spatial corruption, but are very expensive.

The ultimate goal of memory corruption is to control what instructions get executed on the target machine. These are called control flow hijack exploits. Using a temporal or spatial corruption, the attacker gets to control the program pointer and executes a malicious sequence of instructions. Stack Smashing was an early work that made use of a stack overflow vulnerability to inject and run malicious code [53]. As a response to Stack Smashing, a variety of defenses were deployed. Stack Canaries [67] is a strategy of placing a secret value after the return address of the previous function but before everything else of the current function's frame. A stack overflow would overwrite this secret value and thus can be detected. StackGuard [22] is a patch to gcc that implemented Stack Canaries. To prevent code-injections, Data Execution Prevention [8] was developed. DEP prevented execution of any page that contains user provided data. Attackers subsequently developed code-reuse attacks that do not require code-injection. One of the first such attacks is return-to-libc [49], which chains functions contained in libc to create a complex attack, bypassing DEP. In general, these code-reuse exploits that chain together existing code fragments are called Return Oriented Programming(ROP) [19, 63]. ROP chains not only functions, but also smaller code fragments called gadgets in the existing code space of a target machine. ROP attacks are known to be Turing complete.

After the introduction of ROP, there were a series of developments in both attacking and mitigation landscapes. Jump-Oriented Programming (JOP) [64] also chains a series of code blocks called gadgets to create arbitrarily complex attacks. Transferring from one block to another is however done using indirect jumps in contrast to ROP which uses return instructions. Counterfeit Object-oriented Programming (COOP) [59] is another ROP that induces Turing complete attacks by chaining C++ virtual functions. Just-In-Time-ROP (JIT-ROP) [64] leverage special properties of Just-In-Time compiled code to bypass some mitigations for ROP which will be discussed below.

In the previous two paragraphs, we discussed attacks that target to overwrite control flow data and take over the control flow of a program. Another class of attacks is one where attackers target non-control flow data [20]. One of the seminar works in this area is Data-orietned Programming (DOP) [32]. DOP constructs complex data-oriented exploits by chaining together gadgets in the program space. The gadget chains in DOP follow valid CFG paths in contrast to those in ROP. Block-oriented Programming (BOP) [**?**] is similar to DOP, but a gadget is a single basic block in BOP.

Control-flow hijack attacks ultimately hope to induce execution of a code path that does not exist in the original CFG of the program. They achieve this by overwriting control-flow data. Control-flow Integrity (CFI) [5] is one class of the major mitigation techniques that are widely deployed against such attacks. CFI computes the CFG of a program at compile time and uses the constructed CFG to restrict execution flow at run time. To provide backward edge protection, solutions like shadow stacks [24] and stack canaries [67] are used. As CFI needs to construct CFG using static analysis, increasing the precision of the static analysis increases the accuracy of the CFI. Since more precise static analysis degrades performance, engineers often settle for a compromise between performance and precision. CFI through labelling [15, 56, 70, 51, 23, 52] is a popular approach for doing this. Unfortunately, several works have shown CFI is not impregnable from attacks. Works like CFI-Jujutsu [27] have shown that even fine grained implementations of CFI can be bypassed. CFI-bending [18] goes farther

than that and shows even ideal implementations of CGI can be bypassed.

Another class of mitigation techniques against control-flow hijacks is randomization. Some of the earliest work in this regard was in preventing code injections by randomizing the instruction set [38, 11]. With the advent of code-reuse attacks, defenders engineered randomization techniques known as Address Space Layout Randomization (ASLR) [55, 14, 12]. ASLR randomizes the start address of libraries in a program space so that the attacker will not be able to easily predict the locations for finding gadgets. ASLP and ILR are forms of ASLR that generate a high amount of randomness for a small overhead [39, 31].

Manually constructing attacks is a tedious process. Thus a wide array of techniques for automatically constructing exploits have been invented. Fuzzing is one example of these techniques [42, 45]. Fuzzing tries random inputs until a program can be taken to a desired state. Modern fuzzing tools like American Fuzzy Lop (AFL) [69] optimize by reducing the randomness of the inputs. Symbolic execution is an improvement over fuzzing that can reason about all possible inputs to a program [61]. A major problem in symbolic execution is path explosion. There is a significant amount of work in the literature for addressing this. Many of these rely on mixing concolic execution with symbolic execution to prune the space tree. One common solution is called Dynamic Symbolic Execution (DSE) [30] has a concolic execution drive the symbolic execution. Selective Symbolic Execution (S^2E) [21] switches from concrete to symbolic and back from symbolic to concrete depending on various parameters. Automatic Exploit Generation (AEG) [10] is a tool for automatically generating end-to-end control-flow hijacking exploits. Block Oriented Programming Compiler (BOPC) [35] allows automatically constructing chains of blocks for executing an arbitrary set of instructions. Q [60] is a tool for automatically generating end-to-end ROP payload.

Chapter 8

Conclusion

In this thesis, we designed and implemented ACLEG , an automatic CLA exploit generation tool. We tested our tool extensively on the toy examples as well as commercial programs like Firefox and Fuchsia. We developed novel varieties of symbolic execution and reaching definition analysis techniques that are scalable and performant. Even though our tool now is able to generate only double free exploits, the path towards expanding this to currently known CLA exploits and others that might be discovered in the future is clear. Due to limitations in the underlying binary analysis framework we rely on as well as the shortness of time, we were unable to achieve an analysis as complete as we desired. However, the results we have obtained show the significant potential for future research in the field. Improvements in the underlying framework and addressing the issue of tracking symbolic execution through the PLT would specifically be immensely helpful for generating a complete set of CLA exploits possible against a binary program.

Bibliography

[1] Cgo. https://pkg.co.dev/cmd/cgo.

[2] Foreign function interface. https://doc.rust-lang.org/nomicon/ffi.html.

[3] Go-git. https://github.com/go-git/go-git.

[4] mprotect. https://man7.org/linux/man-pages/man2/mprotect.2.html.

[5] Martin Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity: Principles, implementations, and applications. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.

[6] P. Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security'10*.

[7] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium, vol. 10*, 2009.

[8] S. Anderson and V. Abella. Data execution prevention, changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies. 2004.

[9] T. M. Austin, S. E. Breach, and G.S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming Language Design and Implementation*, pages 290–301.

[10] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. Aeg: Automatic exploit generation. In *Network and Distributed System Security Symposium*, Feb, 2011.

[11] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanović. Randomized instruction set emulation. In *ACM Transactions on Information and System Security (TISSEC), vol. 8, no. 1*, pages 3–40, 2005.

[12] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *USENIX Security symposium, vol. 12,*, pages 291–301, 2013.

[13] Bruno Bierbaumer, Julian Kirsch, Thomas Kittel, Aurelien Francillions, and Apostolis Zarras. Smashing the stack protector for fun and profit. 2018.

[14] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 268–279, 2015.

[15] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. Mitigating code-reuse attacks with control-flow locking. In *Annual Computer Security Applications Conference (ACSAC). New York, New York, USA*, 2011.

[16] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, , and M. Payer. Control-flow integrity: Precision, security, and performance. In *ACM COmputing Surveys (CSUR), vol. 50, no. 1*, pages 1–33, 2017.

[17] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities.

[18] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Rockjit: Securing just-in-time compilation using modular control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[19] S. Checkowa, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *ACM Conference on Computer and Communications Security (CCS '10)*, 2010.

[20] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium, vol. 5,*, 2005.

[21] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The s2e platform: Design, implementation, and applications. In *ACM Trans. on Computer Systems (TOCS) 30*, pages 1–49, 2012.

[22] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stack-guard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security'98*.

[23] John Criswell, Nathan Dautenhahn, and Vikram Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *2014 IEEE Symposium on Security and Privacy*, 2014.

[24] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nurenberger, and Ahamd-Reza Sadeghi. Mocfi: A framework to mitigate control-flow attacks on smartphones. In *Symposium on Network and Distributed System Security (NDSS)*, 2012.

[25] J. Devietti, C. Blundell, M. M. Martin, and S. Zdancewic. Hardbound: architectural support for spatial safety of the c programming language. In *ACM SIGOPS Operating Systems Review, vol. 42, no. 2*, page 103–114, 2008.

[26] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the point (er): On the effectiveness of code pointer integrity. In *IEEE Symposium on Security and Privacy*, pages 781–796, 2015.

[27] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA*, August 12-14, 2015.

[28] A. Follner, A. Bartel, Y. Chang H. Peng, K. Ispoglou, M. Payer, and E. Bodden. Pshape: Automatically combining gadgets for arbitrary method execution. 2016.

[29] Ronald Gil, Hamed Okhravi, and Howard Shrobe. There's a hole in the bottom of the c: On the effectiveness of allocation protection. 2018.

[30] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Impl. (PLDI'05)*, pages 213–223, 2005.

[31] J. Hiser, A. Nguyen-Tuong, M. Hall M. Co, and J. W. Davidson. Ilr: Where'd my gadgets go? In *2012 IEEE Symposium on Security and Privacy*, pages 571–585. IEEE, 2012.

[32] H. Hu, S. Shinde, S. Adrian, L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *IEEE Symposium on Security and Privacy (SP)*, pages 969–986, 2016.

[33] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986, 2016.

[34] huku and argp. Exploiting vlc: A case study on jemalloc heap overflows. Phrack, vol. 14, no. 68.

[35] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer. Block oriented programming: Automating data-only attacks. In *in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1868–1882.

[36] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of c. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[37] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Auto. and Algo. Debugging '97*.

[38] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280, 2003.

[39] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 339–348. IEEE, 2006.

[40] C. Lattner and V. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI'05*.

[41] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Tae soo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, February, 2015.

[42] Sutton. M, Greene A, and Amini P. Fuzzing: brute force vulnerability discovery. 2007.

[43] N. D. Matsakis and F. S. Klock. The rust language. In *ACM SIGAda Ada Letters, vol. 34, no. 3*, pages 103–104, 2014.

[44] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. Cross-language attacks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'22), San Diego, CA*, 2022.

[45] B. P. Miller, L. Fredriksen, and B. S. An empirical study of the reliability of unix utilities,. In *Communications of the ACM, vol. 33*, pages 32–44, 1990.

[46] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Cets: compiler enforced temporal safety for c. In *Proceedings of the 2010 International Symposium on Memory Management*, pages 31–40.

[47] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–258, 2009.

[48] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimar. Ccured: Type-safe retrofitting of legacy software. In *ACM Transactios on Programming Languags and Systems (TOPLAS), vol. 27, no. 3*, pages 477–526, 2005.

[49] Nergal. The advanced return-into-lib(c) exploits: Pax case study. In *Phrack Magazine, Volume 11, Issue 0x58, File 4 of 14*, 2001.

[50] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI'07*.

[51] Ben Niu and Gang Tan. Modular control-flow integrity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[52] Ben Niu and Gang Tan. Rockjit: Securing just-in-time compilation using modular control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

[53] Aleph One. Smashing the stack for fun and profit. 1996.

[54] M. Papaevripides and E. Athanasopoulour. Exploiting mixed binaries. In *ACM Transactions on Privacy and Security*, pages 1–29, 2021.

[55] T. PaX. Pax address space layout randomization (aslr). In *http://pax.gresecurity.net/docs/aslr.txt*, 2003.

[56] Jannik Pewny and Thorsten Holz. Control-flow restrictor: Compiler-based cfi for ios. In *Annual Computer Security Applications Conference (ACSAC)*, 2013.

[57] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *NDSS, vol. 2004*, pages 159–169, 2004.

[58] F. Schuster, T. Tendyck, C. Liebchen, A.-R. Sadeghi L. Davi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *IEEE Symposium on Security and Privacy*, pages 745–762, 2015.

[59] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, pages 745–762.

[60] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *USENIX Conference on Security (Security '11)*, 2011.

[61] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium of Security and Privacy*, pages 317–331, May 2010.

[62] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX ATC'12*.

[63] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS'07)*, 2007.

[64] Kevin Z. Snow, Fabian Monrose, Lucas Davi, and Alexandra Dmitrienko. On the effectiveness of fine-grained address space layout randomization. 2013.

[65] L. Szekeres, M. Payer, T. Wei, , and D. Song. Sok: Eternal war in memory. In *IEEE Symposium on Security and Privacy*, pages 48–62, 2013.

[66] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning. On the expressiveness of return-into-libc attacks. 2011.

[67] P. Wagle and C. Cowan. Safeguard: Simple stack smash protection for gcc. In *Proceedings of the GCC Developers Summit*, pages 243–255, 2003.

[68] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen. Paricheck: an efficient pointer arithmetic checker for c programs. In *ASIACCS'10*.

[69] M. Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl/.

[70] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity randomization for binary executables. In *IEEE Symposium on Security and Privacy*, 2013.