PRACTICAL INDIRECT CONTROL FLOW ANALYSIS FOR BINARY EXECUTABLES

Haotian Zhang

Supervising Committee: Jiang Ming
Yu Lei
Shirin Nilizadeh
Mohammad A. Islam

Department of Computer Science and Engineering
The University of Texas at Arlington

This dissertation is submitted for the degree of
Doctor of Philosophy

May 2023

Abstract

Resolving indirect control flow is one of the fundamental challenges in binary analysis. Improving the accuracy of the indirect control flow analysis is vital to the binary analysis domain. Many analysis algorithms and security techniques rely on a precise indirect control flow result, such as recursive disassembling, control flow integrity, data-flow analysis, etc. Incorrect or even inaccuracy indirect control flow analysis results can compromise or even break the assumptions of these analyses. This thesis explores this topic from two directions, altering the indirect control flow analysis to make it more suitable for different scenarios and improving the accuracy of indirect control flow analysis with deep learning.

In the first part, we explore the potential trade-off that can be made in debloating scenarios. Static software debloating often requires an accurate indirect control flow result. However, previous works resolve indirect control flow utilizing the address-taken function, which has too many false positives to debloat the program efficiently. During our observation, debloating does not require the individual indirect control flow result but a set of indirect control flow results mixed together. Instead of solving each indirect control flow, we focus on how the target is loaded from memory. The loaded target can be used in any of the following indirect control flows. We build a novel tool with this methodology to debloat the shared library in MIPS firmware.

In the second part, we explore how deep learning can be applied to indirect control flow resolving problems. Unlike text or picture, which has a more straightforward data relation structure, binary is much more complex, especially in the control flow structure. The graph is a natural representation used in the program analysis domain. We utilize the graph neural network in our augmented control flow graph to learn how to predict indirect callees. We translate the indirect callee prediction problem into a graph's edge prediction problem.

Table of contents

Chapter 1

General Introduction

The control flow of a program refers to the order in which the statements of the program are executed. Control flow can be direct or indirect. In direct control flow, the execution of statements proceeds in a linear fashion, following a predetermined order. In contrast, indirect control flow allows for non-linear execution, where the order of execution is determined dynamically at runtime. Indirect control flow can be challenging to analyze and understand, but it is essential for writing efficient and flexible programs.

Indirect control flow can arise in various programming contexts, including function pointers, callbacks, and exception handling. In such situations, the execution of the program can depend on external factors or user input, leading to non-deterministic behavior. As a result, understanding indirect control flow is crucial for writing robust and reliable software.

The resolution of indirect control flow is a central problem in program analysis and optimization. Resolving indirect control flow involves identifying the possible targets of control transfers, such as function calls or exception handlers. This process can be challenging because the possible targets of a control transfer may not be known until runtime.

Several techniques have been developed for resolving indirect control flow, including static analysis, dynamic analysis, and hybrid approaches. Static analysis involves analyzing the program code without executing it, while dynamic analysis involves observing the program's behavior during execution. Hybrid approaches combine both static and dynamic analysis techniques to achieve better accuracy and performance.

In the first part, we explore the potential trade-off that can be made in debloating scenarios. Static software debloating often requires an accurate indirect control flow result. However, previous works resolve indirect control flow utilizing the address-taken function, which has too many false positives to debloat the program efficiently. During our observation, debloating does not require the individual indirect control flow result but a set of indirect control flow results mixed together. Instead of solving each indirect control flow, we focus on how the

target is loaded from memory. The loaded target can be used in any of the following indirect control flows. We build a novel tool with this methodology to debloat the shared library in MIPS firmware.

In the second part, we explore how deep learning can be applied to indirect control flow resolving problems. Unlike text or picture, which has a more straightforward data relation structure, binary is much more complex, especially in the control flow structure. The graph is a natural representation used in the program analysis domain. We utilize the graph neural network in our augmented control flow graph to learn how to predict indirect callees. We translate the indirect callee prediction problem into a graph's edge prediction problem.

Chapter 2

One Size Does Not Fit All: Security Hardening of MIPS Embedded Systems via Static Binary Debloating for Shared Libraries[1]

Haotian Zhang, Mengfei Ren, Yu Lei, and Jiang Ming
University of Texas at Arlington

[1]Used with permission of the Association for Computing Machinery, 2022. Haotian Zhang, Mengfei Ren, Yu Lei, and Jiang Ming. 2022. One size does not fit all: security hardening of MIPS embedded systems via static binary debloating for shared libraries. In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22). Association for Computing Machinery, New York, NY, USA, 255–270. https://doi.org/10.1145/3503222.3507768

Abstract

Embedded systems have become prominent targets for cyberattacks. To exploit firmware's memory corruption vulnerabilities, cybercriminals harvest reusable code gadgets from the large shared library codebase (e.g., uClibc). Unfortunately, unlike their desktop counterparts, embedded systems lack essential computing resources to enforce security hardening techniques. Recently, we have witnessed a surge of software debloating as a new defense mechanism against code-reuse attacks; it erases unused code to significantly diminish the possibilities of constructing reusable gadgets. Because of the single firmware image update style, static library debloating shows promise to fortify embedded systems without compromising performance and forward compatibility. However, static library debloating on stripped binaries (e.g., firmware's shared libraries) is still an enormous challenge.

In this paper, we show that this challenge is not insurmountable for MIPS firmware. We develop a novel system, named *µTrimmer*, to identify and wipe out unused basic blocks from shared libraries' binary code, without causing additional runtime overhead or memory consumption. We propose a new method to identify address-taken blocks/functions, which further help us maintain an inter-procedural control flow graph to conservatively include library code that could be potentially used by firmware. By capturing address access patterns for position-independent code, we circumvent the challenge of determining code-pointer targets and safely eliminate unused code. We run µTrimmer to debloat shared libraries for SPEC CPU2017 benchmarks, popular firmware applications (e.g., Apache, BusyBox, and OpenSSL), and a real-world wireless router firmware image. Our experiments show that not only does µTrimmer deliver functional programs, but also it can cut the exposed code surface and eliminate various reusable code gadgets remarkably. µTrimmer's debloating capability can compete with the static linking results.

2.1 Introduction

Over the past few years, the spotlight has been on the Internet of Things (IoT) market due to the sheer amount being deployed worldwide [58]. With the IoT boom taking place, cyberattacks

on embedded devices, which surged 300% in 2019 [76, 95], are now accelerating at an unprecedented rate. The vulnerabilities in firmware, a class of software that is written to an embedded device to control user applications and various hardware functions, leave embedded systems open to attacks [28, 42, 30]. Although memory corruption vulnerabilities [104] have been around for decades, they also dominate the share of top-rated threats in embedded devices [33]. Besides, return-oriented programming (ROP) techniques enable attackers to chain together short instruction sequences (i.e., code gadgets) already present in the program's memory to bypass the executable-space protection [92].

Firmware developers rely on C and C++ shared libraries (e.g., uClibc [7]) for fast prototyping and development [115]. Due to the "one-size-fits-all" design, although firmware typically requires a small number of library functions, it has to load the entire library code into memory at runtime. For example, libc code are only used 5% on average by a program [86]. Compared with the small codebase of firmware, the large code space of shared libraries provides enough reusable code gadgets to create Turing-complete malicious programs [92]. Embedded devices are known to have limited hardware resources in terms of CPU performance, storage capabilities, and memory size. Besides, as firmware has to interact with a multitude of low-level peripherals, a robust firmware dynamic execution framework is still an open problem [117, 23, 41, 25, 19]. Therefore, these limitations restrict the adoption of expensive ROP countermeasures to secure embedded systems [2, 5].

Recently, software debloating emerges as a new security hardening solution to reduce the attack surface by removing consumer-unwanted features or unused code, generating a large body of literature [60, 88, 86, 54, 100, 15, 9, 90, 84, 46, 3, 83, 24, 47, 17, 85, 114]. In particular, library debloating techniques [86, 3, 83] have demonstrated their security impact by eliminating a large number of reusable code gadgets from shared libraries. Furthermore, they can significantly reduce the amount of code to be analyzed by other security techniques, such as continuous code re-randomization [112] and control-flow integrity schemes [64, 18].

Static library debloating reveals unique benefits to embedded systems. First, it safeguards firmware without incurring additional runtime overhead or memory footprint. Second, unlike PC software, static library debloating does not compromise firmware forward compatibility. Embedded devices typically have no interface for an end-user to install new application packages; instead, the update mechanism is the single firmware image update: users download a new firmware image from the hardware manufacturer and re-flash it to the device. As a result, the post-deployment library debloating does not interfere with the new firmware image. However, existing library debloating approaches rely on a number of assumptions that are not met by embedded systems. Piece-wise [86] and BlankIt [83] rely on application source code and runtime support (e.g., a custom loader or Intel Pin), while Nibbler [3] leverages debug

symbols to identify and erase unreachable functions. In contrast, firmware's source code and unique build toolchains are typically missing, and *most firmware images are stripped from debug information to save space* [30].

In this paper, we move one step forward to explore the static debloating of library binaries on MIPS architecture, which holds a hefty share of the embedded market [8]. We admit that the static analysis of stripped binaries suffers from several long-standing challenges [10, 75, 59, 91] in the general case, such as distinguishing code from data and indirect control flow resolution. Our key observation is that, compared with ARM, MIPS Application Binary Interface (ABI) [105] makes static binary code analysis much easier, though not enough to make detecting unused library code straightforward. MIPS ABI specifies standard conventions in low-level machine code, such as special purposes registers, stack frame organization, function parameter passing, and position-independent code (PIC) metadata. Mainstream compilers all follow MIPS ABI [45, 71].

We develop an input-profiling-agnostic, static debloating system for MIPS stripped binaries, named μ*Trimmer*, to eliminate unused code present in shared libraries. Note that the effects of multi-entry functions and tail call optimization [75] obscure function boundaries, so our debloating granularity is basic blocks rather than functions. Starting from the imported library functions by firmware applications, we explore library interdependencies and maintain an inter-procedural control flow graph (ICFG), which over-approximates all basic blocks that could be potentially used. Statically resolving accurate indirect control flow targets (e.g., calls using pointers and C++ virtual function dispatch) is proven to be an undecidable problem [66, 87]. We circumvent this challenge by proposing a general method to significantly narrow down the possible *address-taken* blocks/functions—their addresses are very likely to be referenced by indirect jump/call instructions. Our approach covers all indirect jump/call cases, such as callback functions, jump tables, C++ virtual function tables, and exceptions. After the ICFG recovery, all basic blocks that are not present in this ICFG can be safely debloated. At last, μTrimmer overwrites unused basic blocks with trapping instructions [29] and delivers thinned libraries.

We build μTrimmer on top of angr [101] and evaluate the efficacy of μTrimmer with a set of experiments. We run μTrimmer to debloat supporting libraries for SPEC CPU2017 benchmarks, popular firmware applications (e.g., Apache, BusyBox, OpenSSL, and Perl), and a real-world wireless router firmware image. We demonstrate that μTrimmer safely removes unused code by running the officially-provided test suite—the debloated program reveals the same results as the original program's executions. For SPEC CPU2017 Integer suite, our security experiments show that μTrimmer can cut the exposed code surface by 53.4% to 79.9% and eliminate various reusable code gadgets by 56.2% to 78.9%. The dead code elimination

Table 2.1 Comparison of representative library debloating approaches.

	Piece-wise [86]	Nibbler [3]	BlankIt [83]	μTrimmer
No Source Code Needed		✓		✓
No Debug Symbols Needed				✓
No Sample Inputs Needed	✓	✓		✓
No Runtime Support	Custom Loader	✓	Intel Pin	✓
Architecture	x86/x86-64	x86-64	x86/x86-64	MIPS/MIPS64
Debloating Granularity	Function	Function	Function	Basic Block
Load-time Slowdown	~20X	0%	~10X	0%
Runtime Slowdown	0%	0%	~18%	0%
Code Reduction Amount	Medium	Medium	Large	Medium

caused by static linking is taken by recent work [86, 3] as an upper bound of library debloating; μTrimmer's debloating capability competes with the static linking results and outperforms piece-wise [86] and Nibbler [3]. In a nutshell, this paper makes the following contributions:

- Unlike PCs and Servers who can afford a myriad of security protections, embedded devices with limited computing resources are sensitive to deploy advanced software hardening techniques. Our research shows that static binary debloating for shared libraries, which incurs *zero* runtime overhead, has distinctive strengths to secure embedded systems.
- CFG construction is the cornerstone of static binary debloating; but without source code or debug symbols, it is known to be a challenging problem. Our study shows that, by taking advantage of MIPS ABI and PIC specifications, we can find a practical solution to circumvent this challenge and safely erase unused code.
- We evaluate μTrimmer's correctness and security impact with large benchmark programs and real-world embedded system applications. μTrimmer's debloating capability is on a par with the static linking's code downsizing results.

Open source. μTrimmer's demo video is available at YouTube. We have released μTrimmer's source code and non-proprietary dataset to facilitate reuse at Zenodo.

2.2 Background & Related Work

In this section, we present the background information needed to understand our work's motivation and novelty. We summarize the existing literature on software debloating. We focus on the recent papers on library debloating because they are the works most germane to our research. Then, we present MIPS's advantage for binary code analysis and the challenge (i.e., indirect control flow) that we aim to overcome.

2.2.1 Code-Reuse Attacks on IoT Devices

The proliferation of the IoT market makes embedded devices a lucrative target for cyber-criminals. For example, critical security vulnerabilities in WiFi routers and smart home devices allow remote attackers to completely take over the device and enter the home network [51, 21, 80, 78, 98]. Among various attacks against the IoT ecosystem, code-reuse attacks [92] can bypass executable-space protection, leading to catastrophic consequences, such as remote code execution. The potential potency of code-reuse attacks hinges on the variety of code gadgets in the victim program's executable memory. As shared libraries are designed to contain the union of API code required by all possible applications, their large codebase is always the best place to harvest different reusable code gadgets [99].

As IoT devices have substantially less computation and storage capability than conventional computers, developers are using a small C standard library (e.g., uClibc) intended for Linux kernel-based OS on embedded/mobile devices. The total size of uClibc is only about 7% of glibc [39]. Even so, uClibc is still a fertile land to search ROP gadgets [4, 116]. Our study shows that only 17.3% of functions in uClibc are used by firmware on average. Bloated shared library code provides adversaries a large code-reuse attacking surface.

2.2.2 Software Debloating

As software is continuously becoming more sophisticated, software debloating is an effective defense to minimize attacking surface. A variety of schemes have been contrived to remove consumer-unwanted features or unused code [60, 88, 86, 54, 100, 15, 9, 90, 84, 46, 3, 83, 24, 47, 17, 85, 114]. The debloating targets range from program binaries [90, 84, 46], Java applications [60, 17], Docker containers [88], mobile/web applications [15, 9, 85], UEFI firmware [24], Bluetooth stacks [114], and shared libraries [86, 3, 83]. Most existing works assume the availability of source code, sample inputs as the usage profile, or runtime support. For example, DECAF [24] attempts to debloat a maximum set of UEFI firmware modules so that the OS can still be successfully booted. It treats finding such a removable set as an optimization problem and approximates the optimal solution via iterative dynamic testing and metaheuristic search. Unfortunately, these approaches are hardly employable to meet our requirement: *an input-profiling-agnostic, static binary debloating technique that incurs no extra runtime overhead or storage costs*.

Shared Library Debloating In the literature, several techniques have been recently proposed to detect and remove unused code from shared libraries [86, 3, 83]. We compare them with our work in Table 2.1. It lists different assumptions (e.g., source code, debug symbols, and

sample inputs), debloating granularity, performance penalty, and the amount of code reduction. Obviously, our work has fewer assumptions. Below, we discuss their strengths and limitations.

Piece-wise [86] This work first performs a large-scale study to report that library code bloating is pervasive. 95% of glibc code is never used on average. Given the source code of the application and its dependent libraries, piece-wise contains two steps: 1) an LLVM pass generates a full-program dependency graph; 2) a custom loader dynamically loads the functions that are present in the dependency graph. The first step adopts inter-procedural static value-flow analysis [103] to resolve indirect code pointer dependencies within a library. The second step masks unused library functions when loading the library, resulting in an extra load-time slowdown ($\sim$20X). In addition to the load-time slowdown, piece-wise works on each application individually, and thus each application has to load its own custom library code. When piece-wise is applied to multiple applications, the union size of all debloated library versions will far outstrip gains from piece-wise's debloating.

Nibbler [3] This work aims to debloat *non-stripped* library binaries and then create reduced versions. By removing unused code from allowable control flows, Nibbler demonstrates the efficiency boost of continuous code re-randomization [112] and control-flow integrity defenses [18]. However, Nibbler still depends on a strong assumption: library binary code is not stripped from debug symbols. Nibbler takes advantage of these additional information to identify function boundaries, construct library function call graphs, and detect address-taken functions that could be targeted by indirect calls. Unfortunately, all program binaries installed on Linux are stripped of symbols by default. Even worse, to further reduce firmware size, many developers take a more aggressive stripping method to remove binary code's section headers; this will frustrate the tool `objdump` used by Nibbler.

BlankIt [83] At the other end of the spectrum, BlankIt only loads the set of library functions needed at a given call site and wipes out all remaining library functions. BlankIt's just-in-time loading strategy requires the application source code and sample inputs to train a decision tree predictor, which predicts the chain of library functions that are expected to occur at a given call site. This predictor will guide BlankIt's demand-driven loading at runtime. Compared with static debloating approaches, BlankIt's aggressive style shows a very high percentage of code reduction, because only a small portion of library functions are visible during any given runtime window. However, the access to application source code, the deployment environment of dynamic binary instrumentation, and the high runtime overhead make BlankIt impractical to resource-limited embedded systems.

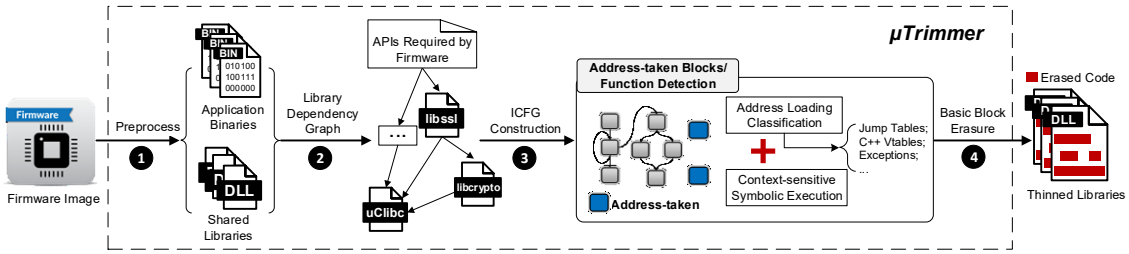

Fig. 2.1 The overview of the μTrimmer. The whole process consists of four steps.

2.2.3 MIPS Architectural Support

As both ARM and MIPS dominate the share of embedded systems, a natural question is whether μTrimmer can work on both architectures. Our work is built on top of two non-trivial pipelines: disassembling binary code and extracting control flow graphs. Our contribution lies in how to identify unused library code without resolving indirect control flow targets. However, the reliability of the initial stage of the pipeline (i.e., code disassembly) actually affects the reliability of the overall approach. The recent study on ARM disassembly tools has demonstrated that two complex problems, which are inline data in code sections and a mixture of ARM, 16-bit Thumb-1, and 32-bit Thumb-2 instruction sets, bring serious challenges to disassembling stripped ARM binaries [59]. In contrast, MIPS binaries do not have such complicated properties, making reliably disassembling MIPS binaries a solved problem.

Furthermore, MIPS Application Binary Interface (ABI) [105] specifications provide handy hints to optimize the address-taken blocks/functions detection. For example, a shared library is position-independent code (PIC), in which most control flow targets are accessed or calculated through the global offset table (GOT) [44]. MIPS ABI specifies two special-purposes registers: 1) $gp register stores the GOT's base address, and 2) $t9 register stores the callee function's address. Monitoring the access to these two registers provides a short cut to explicitly identify the access patterns to the GOT. In contrast, ARM binaries do not have such an advantage—they can use any general-purpose register to calculate and store the GOT indexing. The use of general-purpose registers for address calculations requires us to perform an expensive dada flow analysis (e.g., backward slicing) to achieve the same goal.

2.2.4 Indirect Control Flow

Library debloating requires precise detection of unused code and not missing legitimate code dependencies. We need to keep not only library call chains that are explicitly invoked by firmware but also potential callback functions via pointers. Although MIPS ABI makes static

binary code analysis much easier, constructing a complete CFG is still the biggest obstacle. Failure to identify indirect control flow targets is very likely to incorrectly exclude used code. Previous works have adopted two techniques to mitigate this problem.

Value-set Analysis Balakrishnan and Reps proposed value-set analysis (VSA) technique [10] to identify a tight over-approximation of values in memory slots or registers. VSA is often used to understand the possible targets of an indirect control flow. Redini et al. [90] augment VSA via a new abstract model: signedness-agnostic strided interval. They also apply this new VSA algorithm to binary code debloating, but they only evaluate two very tiny programs with 555 LOC and 192 LOC. The value set obtained by VSA is over-approximated, and its accuracy is subject to the lack of runtime information and path explosion. Therefore, VSA results suffer from a high false positive rate [70]. Our evaluation also demonstrates that VSA is too imprecise for practical binary code debloating.

Address-taken Function Instead of statically resolving indirect control flow targets, a conservative approach is to detect address-taken (AT) functions, whose addresses are referenced as constants somewhere within a module (e.g., executable and shared object). Therefore, they are possible targets of indirect jump/call instructions. Control flow integrity [1] takes all relocation table entries as AT functions. Unfortunately, as all library functions have to be relocated due to PIC, this simple strategy will cause most library code not to be debloated. Nibbler [3] improves the detection strategy by removing AT functions only invoked in unused functions. However, Nibbler's method suffers from two serious limitations: 1) compiler optimization effects can result in arithmetic calculations for function addresses, while Nibbler does not consider such cases; 2) it also misses the complex AT functions caused by C++ virtual functions and read-only global function pointers. As a result, Nibbler may both miss some debloating opportunities and incorrectly remove some used functions.

2.3 Overview

μTrimmer is a sample-input-agnostic, static library debloating technique that works directly on MIPS binaries. The cornerstone of our approach is to construct an inter-procedural control flow graph (ICFG) for each library. Some edges in the ICFG could be missing because we do not attempt to resolve indirect control flow targets. Nevertheless, our address-taken blocks/functions detection ensures that we can find all library basic blocks that could be used. Then, we attach them into the ICFG; that means there are no missing vertices in the ICFG, which is sufficient for the debloating purpose.

Key Insight As the global offset table (GOT) stores relocated addresses, most of the code addressing in position-independent code (PIC) has to rely on reading constant addresses from

the GOT. Library code is PIC as well. Therefore, the vast majority of indirect control flows interact with the GOT: the target address is either directly loaded from the GOT or calculated from a GOT entry. The core of our address-taken blocks/functions detection is to analyze the address loading patterns of the GOT and decide all legitimate addresses that could be referenced. The only exception we observed is using function pointers as read-only global variables; their relocated addresses are stored in the ".data.rel.ro" section. We handle this corner case with a special treatment.

Figure 2.1 illustrates the architecture of μTrimmer. ❶ $\sim$ ❹ represent the following four workflow steps.

1. Preprocess Given a firmware image, we adopt Binwalk [65] to extract the filesystem from the firmware image so that we can obtain application binaries and shared libraries. We also disassemble binary code using the linear scan strategy [75] for the following steps.

2. Library Dependency Graph Then, we collect the APIs required by firmware applications from different sources. As multiple libraries also have inter-module dependencies, the required APIs and the already-extracted libraries are composed to form a library dependency graph. The topological sorting of this graph decides the prioritization of Step 3.

3. ICFG Construction Given the APIs required by its predecessors in the library dependency graph, we construct an ICFG for each library. We categorize different indirect control flows (e.g., jump table and virtual table) according to how they load relocated addresses from the GOT. We apply symbolic execution and capitalize on MIPS ABI and PIC features to determine address-taken blocks/functions for each category. Our fine-grained method significantly narrows down the potential targets and covers all indirect control flow cases, including complicated cases from C++ libraries that cannot be handled by the existing work.

4. Basic Block Erasure The basic blocks that are not included in the ICFG can be safely removed. Our strategy is to simply overwrite these extra basic blocks using a single-byte illegal instruction "0xFF." The benefit of doing so is that any attempt to run the erased code will trigger an exception, and we are immediately aware of implementation errors. Recent binary rewriting works [110, 108, 6, 113, 36, 74] offer an option to decrease the program size as well by deleting unused binary code. Unfortunately, they bear several limitations and trade-offs that can compromise soundness, such as updating code/data references, ignoring computed code pointers, requiring a custom loader to perform runtime address resolution, and non-negligible runtime overhead. We leave it as our future work.

μTrimmer's output is a set of new thinned libraries that can be repackaged into the firmware image. In the following two sections, we present details of our library dependency graph and inter-procedural control flow graph (ICFG) construction.

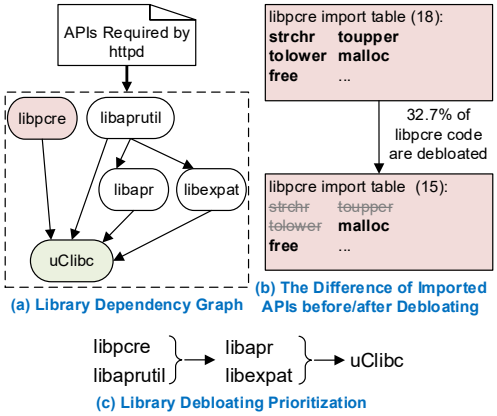

Fig. 2.2 Library dependency graph of httpd (Apache).

2.4 Library Dependency Graph

The starting point of μTrimmer's debloating is to collect the library functions needed by firmware applications. The required functions mainly come from two sources. **First**, we analyze the application binary's import table to obtain imported APIs, which are explicitly invoked by the application. Instead of visiting the import table, a program may also manually load APIs via `dlopen()` and `dlsym()`. In all of our tested programs, we only find one such case for Apache, in which the arguments of `dlopen()` and `dlsym()` are stored in a configuration file. Therefore, we add the APIs defined in the configuration file in the set of required APIs. **Second**, we also include initialization and cleanup routines needed by shared libraries themselves (e.g., `.init`, `.init_array`, and `.fini`) [56]; these functions are defined as arrays of function pointers, which are called by the dynamic linker/loader.

As a library's function may also call the APIs defined in other libraries, given the APIs required by firmware, we need to keep all functions invoked in the library call chain. Therefore, we build a library dependency graph, and its topological ordering schedules which library to be debloated earlier. The root of the library dependency graph is the required APIs by firmware, and leaf node is typically uClibc because other libraries all depend on C standard library.

Figure 2.2(a) shows the library dependency graph of httpd (Apache). We take libpcre as an example. Libpcre depends on uClibc, and the import table of libpcre contains 18 APIs from uClibc. However, not all of these 18 APIs should be kept. After we debloat libpcre library based on the httpd's imported APIs, only 15 uClibc's APIs are still used in the remaining code (Figure 2.2(b)). Libc's `strchr`, `toupper`, and `tolower` are only used by libpcre's `pcre_maketables` function, but httpd does not call `pcre_maketables`. After debloating a library, we collect its imported APIs that are still needed and pass them to the

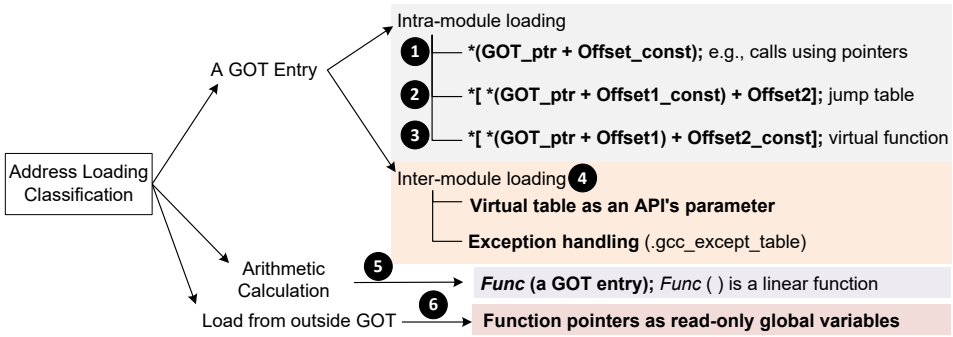

Fig. 2.3 Six address loading patterns according to how indirect control flow targets are loaded from memory.

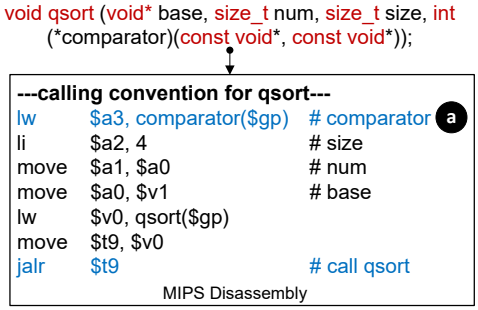

Fig. 2.4 An example of callback function.

successor nodes as the input of ICFG construction. Figure 2.2(c) shows the prioritization of library debloating, which follows the topological ordering of Figure 2.2(a).

2.5 ICFG Construction

At this point, we can use the library dependency graph extracted from the previous step to prioritize the ICFG construction for each library. The input to a library's ICFG construction is the required API list from this library's predecessors. Starting from each required API function's entry point, we construct a control flow graph by detecting basic blocks and connecting the edges between them. All individual CFGs will be composed as a whole ICFG for this library. Please note that we did not differentiate whether an edge is inter-procedural or intra-procedural to determine function boundaries. The reason is that certain compiler optimizations (e.g., multi-entry functions, non-contiguous functions, and tail calls) [75] make transitions between functions implicit.

We mitigate the challenge of statically resolving indirect control flow targets by detecting possible address-taken (AT) blocks/functions. Therefore, the ICFG actually consists of multiple disconnected subgraphs. Unfortunately, the previous works [3, 1] lack a complete picture of

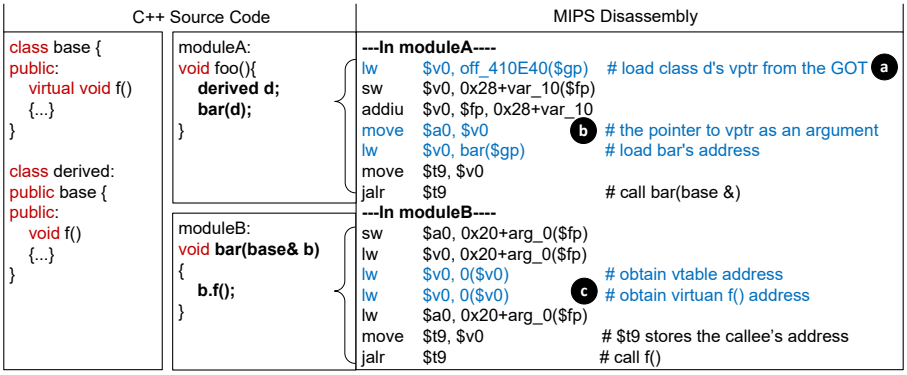

Fig. 2.5 An example of inter-module address loading from libstdc++. We simplify the code snippet for the easy presentation purpose. The moduleA only loads a virtual table's address and passes it as an API's argument. Only the moduleB loads the virtual function's address from the moduleA's virtual table (within the scope of moduleA's GOT).

AT function types, hindering their effectiveness. We present a new, comprehensive taxonomy that covers all types of indirect jumps/calls.

2.5.1 Address Loading Classification

In the PIC code, most indirect control flows have to load constant values from the GOT to recalculate their addresses. Therefore, we classify the address loading patterns according to how indirect control flows interact with the GOT. In Figure 2.3, Type ❶ ∼ Type ❺ either directly load the target address from the GOT or calculate it based on a GOT entry. Our classification significantly narrows the hunting zone for possible AT blocks/functions in a binary file. Now the problem boils down to identifying the GOT's access patterns, thereby producing more tight ICFGs. Besides, MIPS ABI also favors our approach: intra-module access to the GOT has to visit a special-purposes register $gp, which is always used for GOT entry lookup, even if under different compiler optimizations.

Intra-module Loading Each module (executable or shared library) has its own GOT. When the address loading happens within the same module, the most common access pattern is GOT-relative addressing (Type ❶ in Figure 2.3). Function calls using pointers (e.g., callback functions) also belong to this type. As shown in Figure 2.4, when a variable is assigned as a function pointer, the compiler generates instructions to obtain its address via GOT-relative addressing (ⓐ in Figure 2.4). Type ❷ and Type ❸ are corresponding to jump tables and C++ virtual functions, respectively. Both of them occupy a contiguous data area, and they have a similar "pointer to pointer" access pattern. The difference is that the "Offset2" in Type ❷ is a variable because it is decided by the switch-case input. In contrast, the "Offset1" in Type ❸ is

Table 2.2 The distribution of address loading types (see Figure 2.3) in SPEC CPU2017's shared libraries.

	❶	❷	❸	❹	❺	❻
uClibc	84.1%	4.7%	0%	0%	10.5%	0.7%
libstdc++	84.6%	1.6%	3.4%	8.6%	1.8%	0%
libgcc	79.1%	17.1%	0%	0%	3.8%	0%

a variable because a different class has a different virtual table, while a virtual function has a fixed offset in the virtual table.

Inter-module Loading We find that the libraries written in C++ (e.g., libstdc++) may have two complex cases, in which a module's GOT can be accessed by a different module's instructions. Figure 2.5 shows a simplified example of inter-module address loading from libstdc++ library. The moduleA only loads a virtual table's address and passes it as an API's argument (ⓑ in Figure 2.5), but the moduleA has no instructions to load the virtual function address. At runtime, the moduleB will load the virtual function's address from the moduleA's virtual table, which is also within the scope of moduleA's GOT (ⓒ in Figure 2.5). Due to the lack of a global view, the AT function detection in the moduleA does not know which virtual functions are eventually used. We will take a conservative solution to include all possible virtual functions. Another example is C++ exception handling; the real exception handlers' addresses are loaded by a GCC library.

Arithmetic Calculation Compiler optimizations may perform arithmetic on a GOT entry to compute the target address between multiple instructions, hence data-flow analysis is required to detect such a case.

Read-only Global Function Pointers The vast majority of indirect control flow targets come from the GOT. The only counterexample we observed is function pointers used as read-only global variables. They are stored in the ".data.rel.ro" section and initialized to a function's address by the compiler. Our treatment for this case is to label all relocated addresses in the ".data.rel.ro" section as AT functions.

Distribution Table 2.2 shows the distribution of the six address loading types in SPEC CPU2017's shared libraries. Type ❶ is the most common type, but other types also occupy non-negligible portions. Virtual function loading and inter-module loading only happen in libstdc++, and only uClibc uses function pointers as read-only global variables. The portion of used code targeted by each address loading type is analogous to its distribution. Nibbler's AT detection [3] only covers Type ❶ and Type ❷—missing any type could lead to incorrectly removing used code.

2.5.2 Detect AT Blocks/Functions via Symbolic Execution

Our address loading classification guides us to detect address-taken basic blocks and functions using symbolic execution. Given an initial CFG of a library's function, our symbolic execution traverses each CFG node to detect the GOT's access patterns. As $t9 register stores the callee function's address, we also use this value to set the initial state of symbolic execution. Any detected AT blocks/functions are added to our working list, and we perform symbolic execution until no more CFG nodes are discovered. For the most common GOT-relative addressing type (Type ❶), as both $gp and the offset are immediate values, our symbolic execution can easily detect the AT blocks/functions loaded from the GOT. In the following subsections, we discuss how to detect AT blocks/functions related to other address loading types.

Jump Tables

Jump tables are an intra-procedural binary structure to implement switch-case statements. Typically, we mark all target addresses from the jump table as AT basic blocks. When our symbolic execution recognizes the GOT's access pattern like Type ❷, in which "Offset2" is a symbolic variable, we recover the structure used for switch-case control transfer. We adopt a mature jump table recovery method [67, 34, 26], which involves three steps:

1. Identify the jump table's indirect jump.
2. Perform a backward slicing from the indirect jump to calculate the base address and offset range of the jump table.
3. Extract all target addresses from the table.

Note that due to certain compiler optimizations, the extracted target address may not be a valid code address but a constant offset to the GOT. The benefit of doing so is to reduce relocation entries and load-time overhead. When we find such a case, we obtain the real address-taken basic blocks by adding the $gp value to the jump table entries.

Virtual Functions

In Figure 2.5, we have presented the challenge of inter-module address loading caused by using virtual table pointer as an API's argument. Due to the nature of runtime method binding, even if a virtual function invocation happens within the same module, statically resolving all required virtual functions is still an undecidable problem.

In spite of this, a virtual table's creation in PIC is traceable. Each class maintains its own virtual table. When creating a class instance, the program loads a virtual table pointer from the

GOT and saves it in the stack for future use. The type of virtual table pointer loading is either GOT-relative addressing (e.g., ⓐ in Figure 2.5) or via arithmetic calculation, and our symbolic execution can capture both of them. Therefore, we take a conservative strategy to deal with the undecidable virtual function addressing problem:

1. We first find all possible virtual tables stored in a library's binary code.
2. If a virtual table pointer is loaded from the GOT, we treat all virtual functions from this virtual table as AT functions.

In particular, we utilize C++ ABI's definition to identify virtual tables and their scopes in the data section. C++ ABI defines two mandatory fields at the entry of a virtual table: run-time type information (RTTI) and OffsetToTop. RTTI field contains either zero or a pointer to typeinfo; OffsetToTop field contains zero or a negative offset to the primary virtual table. These two mandatory fields are a good indicator to recognize potential virtual tables. After that, our symbolic execution monitors whether a virtual table pointer is loaded from the GOT; if yes, we will label all virtual functions from this virtual table as address-taken functions.

Exception Handling

C++ exception breaks the normal control flow and then executes a pre-registered exception handler. C++ exception handling mechanism relies on both C++ and GCC libraries. The address loading type of the exception handler belongs to inter-module loading—it is loaded from a GCC library. The exception handler information is stored in .gcc_except_table, .eh_frame, and .eh_frame_hdr sections. For C++ binary code (e.g., libstdc++), we parse the exception table and match the connection between try-catch blocks. If our symbolic execution finds a basic block registered in the exception table is used, we will mark the corresponding exception handler code as AT basic blocks.

Listing 2.1 Address loading via arithmetic calculation. The binary code snippet is from uClibc function "re_search_internal."

```
1  la      $v0, loc_10000($gp)
2  //load relocated address of "0x10000" from GOT into $v0
3  nop
4  addiu   $v0, (pop_fail_stack - 0x10000)
5  //(pop_fail_stack - 0x10000) is a compiler-generated constant value; after the add
     operation, $v0 stores the function address of "pop_fail_stack."
6  sw      $v0, 0x1B8+var_8C($sp)
7  //save the function address in the stack for future use
```

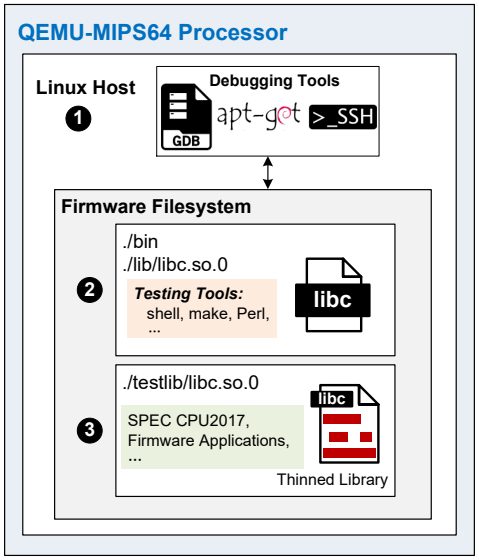

Fig. 2.6 Testing environment setup.

Arithmetic Calculation

In Table 2.2, 10.5% of uClibc's control flow target addresses are calculated from a GOT entry. Listing 2.1 shows such an example from uClibc. Instead of directly loading the function address from the GOT, the program first loads the relocated address of "0x10000" from the GOT. Then, this value is added to an offset to get the relocated address of function "pop_fail_stack." Note that the relocated address of "pop_fail_stack" does not exist anywhere in the binary code. Many control flow integrity methods [1] simply use all relocation table entries as possible indirect control flow targets, and Nibbler [3] only scans load instructions without performing data-flow analysis. Unfortunately, all of them will miss AT functions like "pop_fail_stack." If our library debloating do not consider this address loading type, four debloated benchmarks of SPEC CPU2017 will crash at runtime. Our symbolic execution traces the arithmetic calculation based on the loaded address from the GOT. If the computation result is a valid code address, we will mark this address as a potential reachable target.

2.6 Evaluation

We developed μTrimmer on top of a binary code analysis platform, angr [101]. We reuse angr's disassembly and symbolic execution engine and contribute 3K lines of new code to angr's codebase. Our experiments are performed on a machine with an Intel i9-7900x processor (8-core3.30Ghz) and 16GB memory, running Ubuntu 20.04 LTS.

We evaluate μTrimmer from four dimensions. First, we provide code reduction metrics and demonstrate μTrimmer can deliver functional thinned libraries. Second, we show that μTrimmer's code elimination is very close to the static linking result, which is generally recognized as the optimal library code reduction rate. The third experiment reports ROP gadget reduction results. At last, we debloat a real-world wireless router firmware image to show that μTrimmer can be applied on an entire embedded system.

Table 2.3 The debloating results of SPEC CPU2017 Integer and firmware applications. The data in "Library Statistics" mean the number of libraries each program depends on, the total number of library functions, and the size of library binary code. "#Full" and "#Partial" present the number of functions are fully and partially removed, respectively. "Partial Size" indicates the removed code size from partially-removed functions. "Size" shows the code reduction size in total. All of the size data are in KB. "Time" column lists μTrimmer's running time (hours).

	Library Statistics			Code Reduction Metrics					Time (h)
	#Lib.	#Total Func.	Size	#Full	#Partial	Partial Size	Size	Ratio	
SPEC CPU2017 Integer									
502.gcc_r	1	1,903	519.4	1,488	11	0.3	393.1	75.7%	1.5
505.mcf_r	1	1,903	519.4	1,541	10	0.4	406.2	78.2%	1.2
520.omnetpp_r	3	7,152	1,278.4	4,476	18	0.5	682.7	53.4%	6.2
523.xalancbmk_r	3	7,152	1,278.4	4,599	16	0.3	719.8	56.3%	5.2
525.x264_r	1	1,903	519.4	1,500	10	0.4	390.7	75.2%	1.2
531.deepsjeng_r	3	7,152	1,278.4	4,727	15	0.4	751.7	58.8%	3.8
541.leela_r	3	7,152	1278.4	4,615	15	0.4	736.4	57.6%	5.4
557.xz_r	1	1,903	519.4	1,550	11	0.4	414.9	**79.9%**	1.1
Mean	2	4,527	898.9	3,062	13	0.4	561.9	66.9%	3.2
Firmware Applications									
Apache	5	3,378	945.5	1,421	35	2.5	378.3	40.2%	2.5
BusyBox	1	1,903	519.4	1,092	17	0.8	262.0	50.5%	5.6
Perl	1	1,903	519.4	1,290	16	0.3	345.6	66.5%	2.1
OpenSSL	4	9,400	2,243.3	3,583	38	6.3	607.9	27.1%	5.4
nano	2	2,617	651.7	1,970	13	0.3	444.8	68.3%	1.9
unzip	1	1,903	519.4	1,512	13	0.4	405.1	78.0%	1.7
Mean	2.3	3,517	899.8	1,811	22	1.8	407.3	55.1%	3.2

Environment Setup As shown in Figure 2.6, we build our testing environment using QEMU [14] to emulate a MIPS64 processor. We set up two file system environments in the virtual machine: a generic Linux filesystem and the firmware filesystem. Linux host environment runs debugging tools and the package manager (❶ in Figure 2.6), which are necessary to efficiently develop μTrimmer. All of our experiments, including thinned libraries, run in the firmware filesystem.

Running test suite for empirical correctness evaluation may require additional utility software. For example, running Apache test cases requires `shell`, `make`, and `Perl`, but they also rely on some libc code that is not used by Apache. Therefore, we have to separate the library usage between testing tools (❷ in Figure 2.6) and target programs (❸) by customizing target programs' run-time search path. We found that this intricacy was not addressed by the previous work [86, 3].

Dataset Our test cases include three kinds: SPEC CPU2017 integer benchmarks (four of them are written in C++); popular third-party applications used in router firmware, including Apache, busybox, OpenSSL, Perl, nano, and unzip; a real-world wireless router firmware image, which contains 75 applications and 25 shared libraries. We select SPEC benchmarks because they are often used as complicated evaluation cases for binary code research in the past decade. All target programs are compiled by the optimization level -Os, which is the most common optimization option in embedded systems. We fail to compile two SPEC benchmarks (perlbench and exchange2) due to a known cross-compilation issue [102]. For our router firmware image experiment, we did not have the option to choose any compiler, so μTrimmer works on compiled library code as-is.

2.6.1 Code Reduction and Correctness

Code Reduction Metrics Table 2.3 summarizes the code reduction metrics achieved by μTrimmer on our dataset. The average shared library code reduction ratio for SPEC CPU2017 and firmware applications is 66.9% and 55.1%, respectively. The peak value happens at the data compression benchmark 557.xz_r, in which almost 80% of library code is debloated. SPEC CPU2017 benchmarks depend on three C/C++ libraries, and their per-library debloating results are shown in Table 2.4.

Table 2.4 Per-library code reduction ratio of SPEC CPU 2017 integer benchmarks. "N/A" means this library is not used. "Time" column lists μTrimmer's running time for each library.

	uClibc		libstdc++		libgcc		Overall
	Ratio	Time	Ratio	Time	Ratio	Time	
502.gcc_r	75.7%	1.5 h	N/A	N/A	N/A	N/A	75.7%
505.mcf_r	78.2%	1.2 h	N/A	N/A	N/A	N/A	78.2%
520.omnetpp_r	64.9%	1.8 h	41.4%	4.4 h	72.7%	38 s	53.4%
523.xalancbmk_r	69.5%	0.7 h	42.9%	4.5 h	76.2%	34 s	56.3%
525.x264_r	75.2%	1.2 h	N/A	N/A	N/A	N/A	75.2%
531.deepsjeng_r	72.6%	1.1 h	45.2%	2.7 h	76.4%	39 s	58.8%
541.leela_r	74.0%	1.0 h	41.9%	4.4 h	76.2%	36 s	57.6%
557.xz_r	79.9%	1.1 h	N/A	N/A	N/A	N/A	79.9%
Mean	73.8%	1.2 h	42.9%	4.0 h	75.4%	36.8 s	66.9%

We also report the number of functions and code size that are partially removed. A non-negligible number of functions do not distribute in a continuous code area. Instead, they may have multiple chunks at different areas or even share some blocks with other functions, such as `fdopen`, `fopen`, `fopen64`, and `_authenticate` in uClibc. Table 2.3's data justify our choice of basic blocks as debloating granularity. The last column shows μTrimmer's end-to-end running time, which varies from 1.1 hours to 6.2 hours. Symbolic execution is a well-known

Table 2.5 Per-library debloating capability comparison with static linking for Apache web server. All "Size" data represent the remaining binary code size in KB.

Library	Dynamic	μTrimmer		Static	
	Size	Size	%Redu.	Size	%Redu.
uclibc	519.4	176.8	66.0%	163.2	68.6%
libpcre	80.4	54.1	32.7%	53.9	32.91%
libaprutil	115.0	114.3	0.57%	112.4	2.2%
libapr	131.5	130.3	0.9%	125.3	4.8%
libexpat	99.2	91.7	7.6%	91.4	7.9%
total	945.5	567.2	40.0%	546.2	42.2%

performance bottleneck, and the total running time is positively correlated with the number of libraries.

Empirical Correctness Testing We validate target programs and their debloated versions using officially-provided test suite, and then we compare their execution results to evaluate whether μTrimmer correctly removes only unused code. For SPEC CPU2017, we use the `ref` workload as input, which represents robust correctness testing. The test suite of Apache is collected from the official Apache HTTP Test project [43]. We collect test cases for the remaining programs from their official repositories. All debloated versions pass the correctness testing, and no illegal instruction exception is triggered at runtime.

Value-set Analysis Instead of running our AT blocks/functions detection method, we also tested Redini et al.'s new VSA algorithm [90] to resolve possible targets of an indirect control flow. However, the experiment resul on SPEC CPU2017 is dissatisfactory. Each benchmark has multiple undecidable control flows that have no constraints to limit their targets. If we do not consider such unsolvable control flows, the debloated libraries will incorrectly exclude used code, and none of them can pass our correctness testing.

2.6.2 μTrimmer vs. Static Linking

The effect of static linking represents an upper bound for dead code elimination. However, static linking does not allow memory sharing across processes and may lead to a significantly larger disk footprint, and thus it has been discouraged by many OSs. For example, Solaris removed all static versions of libc in 2004 [40], and Red Hat Enterprise Linux 8 does not support static linking anymore [89]. We compare μTrimmer with static linking to highlight our debloating capability.

As the Apache web server relies on a maximum number of shared libraries in our dataset, we select it for comparison. Table 2.5 shows the per-library debloating results. The second column shows the binary code size of the *original* shared libraries. The third column lists the binary code size of the *thinned* shared libraries. The fifth column shows the binary code size

Table 2.6 The reduction ratio of three common gadget types: syscall, stack pointer update (SPU) and jump-oriented programming (JOP).

Program	%Code Redu.	Total	**Syscall**	SPU	JOP
SPEC CPU2017 Integer					
gcc_r	(75.8%)	76.0%	**79.2%**	75.4%	75.5%
mcf_r	(78.2%)	78.3%	**81.8%**	77.7%	77.8%
omnetpp_r	(53.4%)	56.2%	**77.2%**	53.7%	55.4%
xalancbmk_r	(56.3%)	58.6%	**78.1%**	55.6%	57.8%
x264_r	(75.2%)	76.4%	**81.5%**	76.3%	75.8%
deepsjeng_r	(58.8%)	60.9%	**79.0%**	58.4%	60.2%
leela_r	(57.6%)	58.2%	**79.1%**	55.5%	57.4%
xz_r	(79.9%)	78.9%	**81.9%**	78.4%	78.5%
Mean	(66.9%)	67.9%	**79.7%**	66.4%	68.5%
Firmware Applications					
Apache	(40.2%)	32.1%	**60.1%**	35.3%	30.6%
BusyBox	(50.5%)	51.5%	**62.7%**	54.6%	50.0%
Perl	(66.5%)	67.0%	**68.5%**	67.2%	66.8%
OpenSSL	(27.1%)	25.3%	**71.1%**	28.0%	24.5%
nano	(68.3%)	63.8%	**77.9%**	66.6%	62.6%
unzip	(78.0%)	76.0%	**79.4%**	76.9%	75.5%
Mean	(55.1%)	52.6%	**69.9%**	54.8%	51.7%

after static linking. Note that the Apache web server heavily uses two shared libraries (libapr and libaprutil), thus both μTrimmer and static linking can only remove a small portion of code from them. Overall, static linking removes 42.2% of library binary code, while μTrimmer's code reduction is very close to static linking's result by a small gap of 2.2%. Upon further investigation, we find that the gap of 2.2% is caused by our conservative strategy on handling read-only global function pointers (Type ❻ in Figure 2.3), while static linking can correctly remove functions in the ".data.rel.ro" section.

Directly comparing related library debloating work [86, 3, 83] is infeasible because of their specific assumptions (e.g., source code and runtime support) and different platform requirements. Fortunately, both piece-wise [86] and Nibbler [3] also compare their debloating results with static linking. We measure the difference value of code reduction ratio with static linking as an indirect evaluation. Piece-wise removes 3.9% less code than static linking, and this difference value for Nibbler is 10%. Compared with piece-wise and Nibbler, μTrimmer has fewer assumptions but still outperforms them.

2.6.3 Gadget Reduction

We use ROPgadget [93] to measure three common gadget types: syscall [99], stack pointer update (SPU) [49], and jump-oriented programming (JOP) [16]. Note that MIPS does not contain return instructions and instructions like "call *(memory)," thus conventional gadget types, such as ret-based gadgets [99] and call-oriented programming (COP) gadgets [20], are not available in MIPS. As shown in Table 2.6, μTrimmer seems to be very effective in removing the syscall gadget class, whose reduction ratio is much higher than the respective

Table 2.7 The shared libraries' debloating results for TP-Link Archer A10(V1) firmware. The data in "Library Statistics" mean the number of user-applications requiring this library, the number of other libraries requiring this library, the total number of library functions, and the size of this library binary. "#Full" and "#Partial" represent the number of functions are fully and partially removed, respectively. "Partial Size" indicates the removed code size from partially-removed functions. "Size" shows the code reduction size in total. All of the size data are in KB. "Time" column lists μTrimmer's running time for each library.

Library	Library Statistics				Code Reduction Metrics					Time
	#Bin.	#Lib.	#Total Func.	Size	#Full	#Partial	Partial Size	Size	Ratio	
libcurl.so.4.3.0	2	0	458	225.78	142	1	1.3	77.02	34.1%	9.8 h
libcmm.so	24	0	1,545	653.73	197	17	7.0	78.00	11.9%	32.5 m
libixml.so	1	0	185	71.48	82	0	0	27.36	38.3%	30 s
libupnp.so	1	0	367	227.08	86	7	1.7	55.21	24.3%	9.6 m
libcutil.so	14	1	84	38.91	31	4	0.0	16.53	42.5%	12 s
libthreadutil.so	1	0	83	33.36	58	0	0	27.19	81.5%	2 s
libos.so	22	2	47	8.73	7	0	0	0.91	10.4%	4 s
libiw.so.29	4	0	62	22.91	7	0	0	3.76	16.4%	27 s
libcJSON.so	5	1	66	15.73	20	0	0	2.88	18.3%	33 s
libssl.so.1.0.0	4	1	709	276.36	158	3	0.8	35.06	12.7%	24.3 m
libcrypto.so.1.0.0	4	2	4,563	1,085.77	1,525	31	3.1	215.49	19.8%	3.1 h
librt-0.9.33.2.so	7	1	20	2.03	18	0	0	1.69	83.3%	<1 s
libutil-0.9.33.2.so	1	0	6	1.58	5	0	0	1.22	77.2%	<1 s
libstdc++.so	2	0	2,641	468.13	1,246	2	0.2	182.52	39.0%	5.9 h
libdl-0.9.33.2.so	5	2	10	4.94	2	0	0	0.52	10.5%	4 s
libxml.so	0	1	28	9.53	9	0	0	3.79	39.8%	6 s
libgcc_s.so.1	1	0	1,325	117.88	1,233	1	0.2	82.95	70.4%	1.0 m
libnsl-0.9.33.2.so	1	0	1	0.01	1	0	0	0.01	100.0%	<1 s
liblzo2.so.2.0.0	1	0	151	111.34	117	0	0	73.29	65.8%	1.0 m
libcrypt-0.9.33.2.so	3	0	14	7.08	0	1	0.02	0.02	0.3%	8 s
libz.so.1.2.6	0	0	113	65.16	113	0	0	65.16	100.0%	<1 s
libresolv-0.9.33.2.so	0	3	1	0.01	1	0	0	0.01	100.0%	<1 s
libm-0.9.33.2.so	9	2	166	80.70	127	1	1.5	52.03	64.5%	1.1 m
libpthread-0.9.33.2.so	13	4	214	33.53	77	0	0	8.88	26.5%	1.3 m
libuClibc-0.9.33.2.so	75	23	1,490	315.89	601	8	1.2	115.19	36.5%	1.6 h
Overall			14,349	3,877.65	5,863	75	17.0	1,126.67	29.1%	21.7 h

code reduction. We find that most erased syscall instructions come from unused functions in uClibc. The reduction data for SPU and JOP types are analogous to our achieved code reduction. This experiment indicates that μTrimmer can prohibitively increase adversaries' costs on launching code-reuse attacks.

2.6.4 Firmware Image Debloating

Piece-wise [86] is designed to debloat each application individually, while μTrimmer can be applied on an entire system. We conduct a separate experiment with a real-world embedded device: a wireless router Archer A10 from TP-Link.

This router's firmware image contains 25 shared libraries and 75 applications. The top four libraries in code size are libcrypto, libcmm, libstdc++, and uClibc. The shared libraries libnsl.so and libresolv.so from uClibc are only stubs, containing only one "return" instruction. That is why their executable code size is only 0.01 KB. None of libraries are dynamically loaded via `dlopen()` in this firmware image. Given the whole firmware image as input,

μTrimmer automatically delivers a set of new thinned libraries that can work with all firmware applications. Table 2.7 summarizes the per-library code reduction achieved by μTrimmer and its running time.

μTrimmer first collects the APIs required by 75 firmware applications and generates a library dependency graph. Then, it starts symbolic execution on each library's binary code according to the topological sorting of the library dependency graph. The time data in Table 2.7 represent the processing time for each library, including the time taken to identify unused code. Without the debug symbol information, function recognition in stripped binary code is still an open question [75]. We use IDA Pro's function recovery heuristics [55] to report the number of functions that are fully and partially removed. Next, we report some interesting observations from our experiment.

Two applications are C++ programs, and the others are C programs. C++ programs depend on the bulky libstdc++ library, which is also the only library written in C++. Table 2.7's Column 2 and Column 3 reflect which libraries are commonly depended. Almost all applications and libraries require uClibc. Even so, we can still remove 36.5% of uClibc's executable code. Code bloat is not equally distributed in libraries. In the worst case (libz.so), we found that it was never used by any firmware application so that μTrimmer can remove the entire library. A potential configuration error during the compilation of `cURL` includes libz.so by mistake. In the best case (libcrypto.so), we only removed 0.3% of its code size, indicating most of APIs from libcrypto.so are used.

Four sophisticated libraries (libcurl, libcrypto, libstdc++, and uClibc) occupy 94% of μTrimmer's running time, because their binary code contains a large number of conditional branches. The overall running time of μTrimmer is 21.7 hours. Considering μTrimmer is an automated tool without affecting runtime performance, the overhead is acceptable. After μTrimmer's debloating, we repackage the debloated filesystem back into the firmware image and flash it into a router. We deployed this debloated router device in a university laboratory, and it has run smoothly since September 2020.

2.7 Discussion and Future Work

Library debloating for embedded systems shows promise, but μTrimmer is still in its infancy. This section discusses μTrimmer's limitations and its applicability to other platforms

Static Analysis Limitations Our approach bears similar limitations with static analysis in general. For example, the challenge of disassembling stripped ARM binaries [59] severely limits μTrimmer's adoption in the ARM platform. When a virtual table pointer is loaded from the GOT, we conservatively include all virtual functions from this virtual table. Besides,

the various obfuscation techniques [27, 11] will undoubtedly deter extracting an accurate control flow graph from binary code. However, the firmware running on resource-constrained embedded devices is rarely obfuscated, because code obfuscation can result in a non-negligible performance drop. For manually loaded APIs via `dlopen()` and `dlsym()`, now we can handle them when these two API arguments can be statically decided (e.g., they are hard-coded in binaries). However, the arguments of `dlopen()` and `dlsym()` can also be dynamically generated, and μTrimmer cannot guarantee the safety of library debloating in this case. Piecewise [86] and Nibbler [3] share the same limitation. One possible mitigation is profiling firmware applications with common workloads to reveal the dynamically generated arguments of `dlopen/dlsym`.

Applicability to Other Architectures Binary disassembly concerns aside, our proposed technique is a general approach to identify reachable code for position-independent code by monitoring GOT's access patterns rather than statically solving each indirect control flow. We want to clarify that our approach is not tied to a specific ABI. Utilizing MIPS ABI provides hints to explicitly identify the access operations to the GOT and thus optimizes the address-taken blocks/functions detection, but the overall methodology to detect unused library code also applies to other architectures, including ARM, X86, and RISC-V. For example, X86 may use a stub call to get the current function address instead of $t9 register in MIPS, and we can handle it with small engineering efforts.

No Soundness Guarantee We empirically learned the address loading patterns for indirect control flows, and they are general to the PIC of other ISAs. We did not claim the soundness of our approach because in the binary code analysis domain, theoretical guarantees can be weakened by the toolchain. For example, the underlying binary code symbolic execution is heuristics-based and not sound. We empirically evaluated the correctness by running the officially-provided test suite, which is common for most software debloating papers.

Security Evaluation Metrics Debloating techniques using ROP reduction as security metrics has caused controversy, because only removing unwanted features or unused code cannot prevent code-reuse attacks entirely. Skilled adversaries can still search available ROP gadgets from the remaining codebase, albeit in a more limited form. As μTrimmer achieves the goal of removing code involved in allowable control flows with *zero* runtime performance penalty, a possible enhancement is to combine μTrimmer with continuous code re-randomization and control-flow integrity techniques; Nibbler [3] has demonstrated such a combination is a promising direction.

Delete Unused Code Embedded devices have limited computation resources, and firmware images form a closed software world. These characteristics make static debloating particularly attractive. Currently, we rewrite unused code with illegal instructions to help us quickly locate

any implementation errors. Our correctness testing has demonstrated μTrimmer's result is reliable. Deleting unused code from library binaries also benefits embedded systems, because less shared library code size means better instruction cache performance and faster loading time. One of the key challenges in binary rewriting is to update code pointers and data references, which requires resolving indirect control flow targets accurately. This undecidable problem is also what we try to circumvent in our paper. Currently, no tools can guarantee a successful binary rewriting in practice [110, 108, 6, 113, 36, 74]. Therefore, our future work is to explore binary rewriting to achieve the goal of code size reduction.

2.8 Conclusion

Static binary debloating for shared libraries is a promising but also challenging research task; it can significantly reduce the code-reuse attacking surface without incurring extra performance or storage costs. In this paper, we demonstrate that static library debloating is not an insurmountable obstacle on MIPS architecture. The potential customers of our proposed solution are individuals and companies who want to secure embedded systems via automated binary hardening [50].

Chapter 3

Indirect Call recovery using Augmented Control Flow Graphs with GNN

3.1 Background & Related Work

3.1.1 Graph neural network

Graphs are widely used to model complex data structures such as molecular structures, social networks [48, 94, 77, 32, 22], or control flow graph, data flow graph, abstract syntax tree in the field of program analysis. In contrast to traditional data structures such as arrays or matrices, graphs are characterized by their non-linear and irregular structure. This makes it challenging to apply traditional machine learning techniques to graph data. The basic idea behind GNNs [96] is to propagate information across the graph structure, enabling the model to extract useful features and make predictions.

Typically, GNNs have two phases: the message passing phase and the readout phase. In the message passing phase, information is propagated across the graph structure through a series of local operations. Each node aggregates information from its neighbors, and this information is combined with the node's own features to produce a new feature vector. This process is repeated iteratively until the information has propagated across the entire graph. In the readout phase, the final node representations are used to make predictions. This can be done through a variety of methods, such as averaging the node representations to produce a graph-level representation, or using a graph-level classification model to classify the entire graph.

Heterogeneous graph

A heterogeneous graph is a graph that contains nodes and edges of multiple types. This means that the nodes and edges in the graph represent different types of entities and relationships, rather than a uniform set of entities and relationships. An illustration of a heterogeneous graph can be observed when a graph is required to represent multiple types of information, such as code and data, or control flow and reference relationships. Recent research in heterogeneous graph analysis [97] has focused on developing new techniques for graph embedding, which aim to map the nodes in the graph to a low-dimensional space while preserving the graph structure and the heterogeneity of the nodes and edges [52]. This can be achieved through the use of heterogeneous graph neural networks (HGNNs), which extend the traditional graph neural network models to handle heterogeneous graphs.

Graph Positional Encoding

Graph positional encoding (GPE) is a technique used in graph neural networks (GNNs) to incorporate the position information of nodes in a graph. The basic idea behind GPE is to assign a unique position vector to each node in the graph, which captures its relative position with respect to other nodes in the graph [38, 37]. By incorporating position information into the node representations, GPE can help the model better understand the structure of the graph and capture the local and global relationships between nodes.

3.1.2 Deep learning in binary analysis

Traditionally, binary analysis has been performed using static and dynamic analysis techniques. However, these methods have limitations in their ability to handle the complexity and diversity of programs. Deep learning, on the other hand, has shown promising results in addressing these challenges by providing a flexible and scalable approach to binary analysis. Many different tasks in binary analysis are covered by deep learning, such as debug information revcovery(Debin [53]), binary similarity detection(Trex [81]), type recovery(Stateformer [82]), function name prediction(SymLM [61], NERO [31]). However, their approach only focus on the assembly code itself and also suffer limitations of natural language processing domain.

Natural language processing

For binary program, it can be represented as the machine code or assembly language. Both of them are hard to be used in machine learning. Word embedding from Natural language processing comes handy. An embedding is a method that can translate high-dimension text into

low-dimension vectors and groups similar properties of different instructions. BERT [62] is the state-of-the-art model for word embeddings. There are also many works which have been done to embed the binary programs, such as Instruction2Vec [68], InnerEye [119], Asm2Vec [35], and PalmTree [69]. PalmTree is a general-purpose instruction embedding model based on the BERT model and outperforms the other models. It is a self-supervised model and capable capture the relationship between instructions. In our work, we apply PalmTree for instruction embedding processing.

3.1.3 Information loss

There are several information loss pitfalls in binary from the previous approaches, notably data information, out of vocabulary issue and path explosion.

Data vs Code

Many previous works [81, 82, 61] only use assembly information to present a binary file used in machine learning. However, the data section of a binary file contains additional vital information such as function pointers, jump table targets, initial variable values, and string information. Function pointers and jump table targets are crucial for constructing control flow while the initial variable values and string information are useful for type-based matching analysis. Regrettably, these significant pieces of information have been neglected in previous works.

Out of vocabulary

The vocabulary of a language model typically consists of a fixed set of words that have been pre-defined during the training process. This set of words is typically limited to the most frequent words in the training data. Out of vocabulary (OOV) is a common problem in natural language processing (NLP) and refers to words that are not included in a given language model's vocabulary. OOV words can be problematic as they can result in errors or inaccuracies in NLP tasks. Address and symbols are OOV examples in binary analysis scenarios. One of the common approach is replacing uncommon number or symbols with a special token, such as [num] or [sym] [68, 69]. Unlike the approach only encode numbers or symbols into one single token, Callee [118] tries to encode unlimited number or symbols into a fixed number of symbols, they set the hyperparameter to 10 in their work. However, both of these don't solve the problem, their are still dramatic information lost during this embedding process.

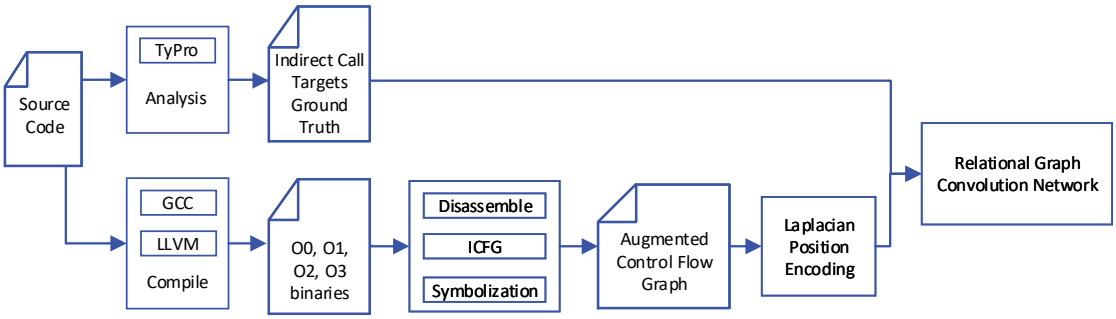

Fig. 3.1 The overview of this indirect call prediction work.

Path explosion

The input of NLP models typically assumes a linear relationship among the inputs, which results in the loss of critical control flow information. Existing approaches, as described in [118, 81, 82, 61], typically rely on linear inputs that either represent the function's semantic regarding paths or just the address order. However, it is challenging to represent a graph using a limited number of paths.

3.2 Overview

3.2.1 Key insight

As mentioned in Section 3.1.3, encoding and learning arbitrary numbers using NLP embedding is a challenging task for deep learning models. However, in the context of binary analysis, unique large numbers, such as addresses, share a special property of being references. These references can be easily represented as edges in a graph. To better represent the program, we augmented the control flow graph with reference edges to capture these important relationships.

To incorporate data information, we introduce a data node into the control flow graph and connect it with the reference edges. This allows us to capture not only the control flow but also the data flow of the program.

By representing the program as a graph, the issue of path explosion is also resolved, as we no longer need to constrain the input to a linear representation. Therefore, graph-based representations provide a more comprehensive and effective approach to representing programs for binary analysis.

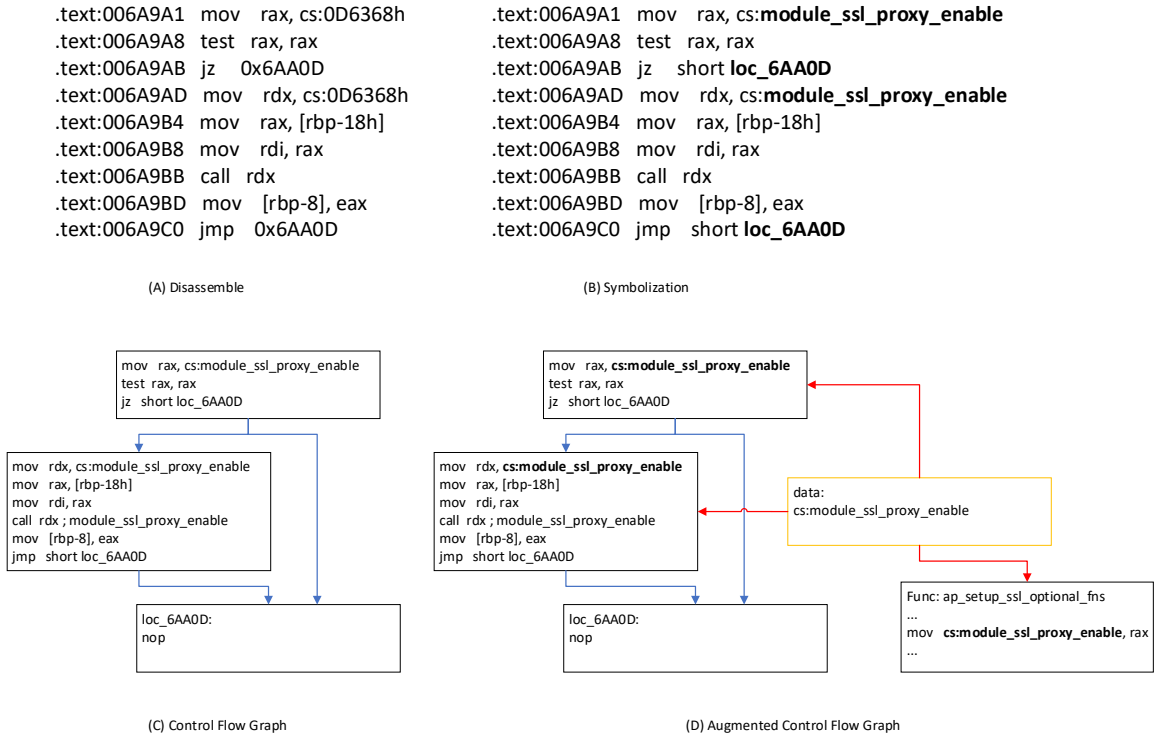

Fig. 3.2 Data processing example of this work. This code piece is picked from apache web server ap_ssl_bind_outgoing function.

Figure 3.1 is the overview of this work. We will cover how we collecting source code data, preprocessing binaries, RGCN details in the next chapter.

In figure 3.2, (A) is the partial disassemble result of the apache web server's ap_ssl_bind_outgoing function. As we can notice the data is loaded at cs:0D6368h data section and the jump target is 0x6AA0D. To avoid the OOV problems, we can not encode any address as we want. During the assembly encoding process, the numeric data will be replaced by a special token, such as [data] [82, 81]. The connection between the address is lost. (B) is the symbolization process of the disassembly code. The address of the data section and code section is represented as special edges in graph. In the following process, we will distinguish reference source and target by assigning different types of edges, which will give a strong hint for a relation reference. (C) is the control flow graph without solving the indirect call. (D) is the augmented control flow graph. A data node is added into the graph and as you can see, from the new data reference edges, a new connection to ap_setup_ssl_optional_fns functions is revealed. This function is the setup function to prepare the indirect callees.

3.3 Model

3.3.1 Dataset and Ground truth collection

There is no "ground truth" in the world of indirect control flow resolving. Dynamic methods like Intel PT or QEMU simulator tracking can collect execution traces. However, due to the limit of the input, dynamic methods can not guarantee coverage or false negative. Static methods, on the other hand, currently can not avoid false positives. Nowadays, the "state-of-practice" method is Clang-CFI, which already applies to many real-world applications, such as Linux Kernal. However, it is still imperfect, as it can not guarantee either soundness or completeness. Two following works are trying to complement the limitations of the Clang-CFI. TypeDive [72], which implements the Multi-Layer Type Analysis, uses additional type hierarchy structure information to further refine the valid target of each indirect call. Successfully reduced the false positives of the Clang-CFI. While instead of directly matching the type information, Typro [12] suggests collecting type propagation rules can reduce the false negative compared with Clang-CFI.

Information such as type and prototype is lost during the compilation process from the source code to binary. It is less accurate and harder for binary-level analysis than source code-level analysis. We treat the source code-level analysis as the best possible result we can get during binary-level analysis. We aim to train a binary-level indirect control flow resolver as good as the source code level. Both TypeDive and Typro are excellent work analyzing indirect calls at the source code level, which can be used as the ground truth in our deep learning training. We value more on the fewer false negative in indirect control flow resolving. So we choose Typro's analysis result as our ground truth. We apply Typro to analyze the collected project's indirect call and treat it as the ground truth of our binary-level prediction. If the project is failed to be analyzed by Typro (mainly due to can not being compiled by Clang), we drop these projects.

3.3.2 Preprocessing

Symbolization

Symbolization is another open problem and creates massive difficulty in binary rewriting areas [111, 109]. The first paper "reassemble disassembly" [111] tries to deal with the symbolization challenge, suggest if all the symbols can be resolved, the disassembled assembly can be compiled back to a workable binary. Which means without the address information, fully symbolized code and data can fully present the binary. However, as summarized in SoK paper [79], binary rewriting tools and binary analysis tools(including IDA pro), still heavily

rely on heuristics and assumptions, which introduce the false positive and false negative. But the overall precision is 99.92 on average. As the address itself introduce OOV issues in deep learning. Using symbolization as a replacement is an essential decision to conservative most information of the binary.

Then we perform symbolization with these heuristic rules the same as angr.

- Exclude data units that are floating points
- Brute force operands and data units
- Pointers in data have machine size
- Pointers in data or referenced by other xrefs can be non-aligned
- Enlarge boundaries of data regions
- While scanning data regions, use step-length based on type inference

Augment data information into the control flow graph

We introduce the data node type, code-to-code reference edge type, code-to-data reference edge type, data-to-data reference edge type, and data-to-code reference edge type into the control flow graph. A data node is after we symbolize all the to-data references. We divide the data sections with these labels into each data node. For code-to-code reference, the code label is added to the assembly code, and the corresponding edge is handled during the control flow graph construction. For code-to-data reference, we add a data-being-referenced edge from the referenced data node to the referencing basic block node. For data-to-code reference, we add a code-being-referenced edge from a referenced basic block node to the referencing data node. For data-to-data reference, we add a data-being-referenced edge from the referenced data node to the referencing data node.

Position Encoding in GNNs

Traditional message passing GNNs suffer from poor performance when there is a lack of positional information of nodes, especially in tasks such as cycle detection. To address this issue, node positional encoding has been proposed and has shown to effectively improve the performance of many tasks, including social network analysis and molecule analysis. [38]

Node positional encoding annotates the structural position of nodes within a graph. In our work, we apply Laplacian eigenvectors as node positional features in the form of Laplacian Positional Encoding [13]. To achieve this, we first transfer the heterogeneous graph to a

homogeneous graph and apply Laplacian Positional Encoding on the homogeneous graph. Then, we add the node positional encoding back into the heterogeneous graph.

One strength of this method is that Laplacian Positional Encoding is network agnostic, meaning it encodes the position into the node features. However, there are some downsides to this approach, including the fact that it does not consider the relationship between different types of nodes and edges. Additionally, this method has a large overhead on very large graphs. Despite these limitations, the use of Laplacian Positional Encoding has shown promising results in various graph-related tasks.

3.3.3 Heterogeneous Graph Neural Networks

The graph becomes a heterogeneous graph with the introduced node types and edge types. We apply a relational graph convolutional network (R-GCN). Our augmented control flow graph is defined as

$$G = (V, E, R, T)$$

. v_c is the basic block nodes, v_d is the data nodes, where

$$V = \{v_c, v_d\}$$

. We seperate the call edges r_c with the other control flow transfer edges r_t. The r_{cc} denotes code-to-code reference edge. The r_{cd} denotes code-to-data reference edge. The r_{dc} denotes data-to-code reference edge. The r_{dd} denotes data-to-data reference edge.

$$R = \{r_c, r_t, r_{cc}, r_{cd}, r_{dc}, r_{dd}\}$$

. The edge is defined with the source node, relation edge type and destination nodes.

$$(v_i, r, v_j) \in E$$

.

The message passing function defined as

$$h_v^{(l+1)} = f(\sum_{r \in R} \sum_{u \in N_v^r} \frac{1}{c_{v,r}} W_r^{(l)} h_u^{(l)} + W_0^{(l)} h_v^{(l)})$$

. h_u^l is the feature representation at layer l. W_r^l is the weights at layer l. $c_{v,r}$ is the normalized by node degree of the relation.

For predicting whether an indirect call r_c exist between the indirect callsite v_i and a potential callee blocks. f_{r_c} denotes the probability output of the model. h_t denotes a true target callee. h_f denotes a false target. The loss function is defined in two parts for the indirect call edge predictions.

$$l = -log f_{r_c}(h_i, h_t) - log(1 - f_{r_c}(h_i, h_f))$$

We maximize the prediction while minimize the false edges.

3.4 Evaluations

3.4.1 Evaluation Setup

Experiments are performed on a Windows 10 machine with subsystem for Ubuntu 20.04 LTS. The machine has an Intel(R) Xeon(R) Gold 6144 CPU @ 3.50GHz, two NVIDIA Quadro RTX 6000 GPUs and 512GB RAM. The Python 3.6.8 is with dgl 0.9.1 and PyTorch 1.10.2.

Datasets

This study employed a Github crawler [57] to collect the programs used for training and testing purposes. In total, 8,533 projects were gathered from Github. The ground truth indirect callsite used for both training and evaluation were generated by applying a source code level CFI method. Specifically, the Typro [12] tool was used for this purpose. Further discussion regarding the collection of ground truth data can be found in Section 3.5.1. After filtering out programs that failed to compile and those that lacked indirect control flow, a dataset of 6,431 programs remained. These programs contained a total of 12,267,610 valid indirect call targets, which were used for training and evaluation purposes. For each program, negative examples were selected online. This involved randomly selecting the same number of functions that were not part of the calling target. For instance, if a program contained 100 functions and one of the indirect calls had four potential targets (Func01, Func03, Func22, Func44), during training or testing, four functions were randomly selected as negative examples, such as (Func02, Func15, Func28, Func35). This approach ensured that the training and evaluation datasets had a balanced number of positive and negative examples, while also introducing a degree of variability. By employing this methodology, the training and evaluation datasets were optimized for machine learning purposes, thus improving the overall performance and accuracy of the models. We randomly split the dataset into three set. 80% for training, 10% for validation and 10% for testing. Callee claims they have severe overfiting issues when only splitting binaries into different groups, F1 drops sharply to 53.7%. So they randomize the pairs

across binaries, which means the model has seen the callsite in training data, then gives the final result. [118] We doubt this approach disclose the encoding of the callsite in testing , which can not measure the model's generalibility. We only randomlize binaries, and ensure all the callsite and callee in other set is never seen in training set.

The dataset was randomly divided into three sets, with 80% for training, 10% for validation, and 10% for testing. However, there were concerns regarding overfitting issues in the training process when the binaries were split into different groups. This led to a significant drop in the F1 score, down to 53.7% [118]. To address this issue, they introduce where the pairs were randomized across the binaries. In their approach, the model had already seen the callsite in the training data, which could result in the encoding of the callsite being disclosed for testing. As a result, the generalizability of the model could not be accurately measured. To overcome this issue, we only randomize binaries, while ensuring that all callsites and callees in other sets were not present in the training set. This approach allowed for the development of a more robust and accurate model with improved generalizability.

Models and hyperparameters

The GNN model utilized in this study is a three-layer RGCN [97], with a three linear layer link predictor. The model was trained using DGL front-end [107] on top of PyTorch. During training, the input to each batch was a program's graph, which was learned and tested across all the ground truth pairs of that program. The network was trained for 10 epochs with a learning rate of 0.001, and a hidden layer feature of 512. A dropout rate of 0.2 was also utilized to prevent overfitting.

Evaluation metrics

To measure the performance of the model, we utilized commonly used metrics, including Precision, Recall, F1-Score. These metrics were calculated based on the number of True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN) generated by the model's predictions. Precision is the ratio of TP to the sum of TP and FP, which measures the model's accuracy in identifying only the relevant results. Recall is the ratio of TP to the sum of TP and FN, which measures the model's ability to identify all relevant results. F1-Score is the harmonic mean of Precision and Recall, providing a balanced measure of both. These metrics are commonly used to evaluate a model's overall performance and allow for a more nuanced understanding of the strengths and weaknesses of the model. In addition to Precision, Recall, and F1-Score, we also report the Area Under the Receiver Operating Characteristic Curve (AUROC) metric as a measure of the model's performance. AUROC is a commonly

used metric for binary classification models, and it measures the model's ability to distinguish between positive and negative examples across a range of classification thresholds.

3.4.2 Performance

Overall performance

As shown in Setting 0 in Table 3.1, our model has an F1 of 94.25%, precision of 93.59%, Recall of 94.95% and AUROC of 96.42%.

Ablation Studies

In order to evaluate the effectiveness of our proposed approach, we compare our model in different settings by turning off specific features and analyzing their impact on performance. Specifically, we investigate the effects of "Revedges," which determines whether we generate reverse edges in our graph, "Data node," which determines whether we add the data node in the graph, and "Func node," which determines whether we add a virtual node for each function that connects all the function's basic block nodes. We also evaluate the impact of "Ref Edges", which correspond to adding reference edges for data and code references. Then, we analyze the impact of treating call edges as a different type of edge than normal basic block edges. Finally, we add the position encoding into each node to see if the position information will improve the accuracy. We refer to these different settings as Setting 1 through Setting 6, respectively. By comparing our model's performance across these different settings, we can determine the relative importance of each feature and how it contributes to the overall effectiveness of our proposed approach.

We have made several observations regarding the performance of our proposed model, as follows:

Firstly, we found that the presence of reverse edges is crucial in our Setting 1. As the control flow graph and references are directed graphs, the propagation of information is limited in one direction without the use of reverse edges. To address this, we added new reverse edge types for each different edge type, allowing for backward propagation of execution traces to be learned separately from forward execution traces.

Secondly, we observed in Setting 2 that the inclusion of data information is important for resolving indirect calls. Upon removal of the data node, the performance of the model significantly decreased.

Thirdly, we found that the addition of a virtual function node and separation of call edge types had little impact on the model's performance in both Setting 3 and Setting 5. This can be

considered as a good trade-off to reduce the model's parameters, which would result in lower hardware requirements.

Fourthly, we determined in Setting 4 that references are another crucial component. With the aid of reference information, the model was able to achieve better results.

Lastly, the Laplacian Positional Encoding has bad influence in our testing. Most may due to not capture the type information and relationships in the heterogeneous graph.

Transfer learning on direct calls

In this work, we adopt the direct call pretraining method proposed in Callee for predicting indirect calls. This method involves training the model first on direct call targets and then transferring the learned parameters to predict indirect call targets. However, we observed that this approach did not lead to significant improvement in the results(Setting 8 in Table 3.1), as reported in Callee. We speculate that there are two possible reasons for this. Firstly, we already have a sufficiently large dataset for indirect calls, and additional learning material may not be necessary. Secondly, the nature of direct calls is different from that of indirect calls. While direct calls have matching argument information between the caller and the callee, the negative examples sampled from functions that are not direct call targets can violate this principle and introduce harmful biases to the model [73]. These factors may have limited the effectiveness of the direct call pretraining approach in our experiments.

Comparision

We compare our model with the state-of-the-art approach Callee. Since it is not open source yet. We use their report result for comparasion. Their best F1 score is 94.6%. We achieve a very comparable result of F1 score of 94.25% without the leaked testing dataset concern (stated in 3.4.1). BPA reports precision of 57.6% and recall of 100%, the F1 score is 73.1% [63]. TypeArmor precision of 35.1% and recall of 99.9%, the F1 score is 51.9% [106].

Time efficiency

The most time consuming part is generating control graph and inquiry asembly embedding from palmtree. The average time to generate a program's argumented control flow graph with embedding takes 73.98s. The average inference time for a program in our GNN model is 0.17s.

Table 3.1 Ablation Studies on Model performance. "Revedges" refer to whether add reverse edges in graph. "Data node" and "Func node" refer to add data or function node in graph. "Ref Edges" refer to whether add the reference edges in graph. "Call Edges" refer to whether assigning separated call edges type or replacing call edges with normal control flow edges. "Position Encoding" refer to whether add position encoding for each node.

Setting	Revedges	Data node	Func node	Ref Edges	Call Edges	Position Encoding	Transfer Learning	Test set Evaluation			
								F1	Precision	Recall	AUROC
0	True	True	True	True	True	False	False	94.25%	93.59%	94.95%	96.42%
1	False	True	True	True	True	False	False	90.41%	91.66%	89.19%	94.85%
2	True	False	True	True	True	False	False	92.00%	93.19%	90.83%	96.04%
3	True	True	False	True	True	False	False	93.80%	92.32%	95.33%	96.44%
4	True	True	True	False	True	False	False	91.10%	92.05%	90.17%	94.28%
5	True	True	True	True	False	False	False	93.78%	91.93%	95.71%	95.25%
6	True	True	True	True	True	True	False	93.58%	91.56%	95.70%	96.32%
7	False	False	False	False	False	False	False	89.61%	89.20%	90.01%	90.45%
8	True	True	True	True	True	False	True	93.91%	93.28%	94.55%	96.11%

3.5 Discussion and Limitations

3.5.1 Ground truth collection

Collecting ground truth for indirect control flow is a challenging task, and there is no perfect solution to obtain accurate target sets. Generally, there are two ways to collect ground truth data. One way is to use dynamic analysis approaches, which involve recording execution traces using software or hardware methods such as Intel Processor Tracing. The ground truth data collected using this approach is 100% accurate, but it may not cover all possible targets, resulting in potential false negative ground truth data. Another method for collecting ground truth is through static analysis. While static analysis may enlarge the set of indirect call targets, it is still relative accurate at source code level. Therefore, we treat the state-of-the-art source code level analysis results as our ground truth data, which we use for training and evaluating our model. Although the ground truth data obtained through static analysis may not be 100% accurate, we believe it provides a sufficient and reliable representation of the indirect call targets for our purposes.

3.5.2 Indirect jumps

At present, the proposed model has only been trained to recover indirect calls. However, the recovery of indirect jumps poses a significant challenge as there is no trustworthy data available for training purposes. Many Control Flow Integrity (CFI) approaches do not adequately protect against indirect jumps, or only provide coarse-grained protection as discussed in [18]. If the training set problem can be addressed through source code level methods, it is possible that this model could be used to solve the issue of indirect jumps, as the inclusion of data information is crucial in this regard.

3.6 Conclusion

To summarize, in this study we have proposed a novel approach that represents both data and reference information using special nodes and edges in a graph, leading to improved preservation of crucial information and achieving better results compared to previous methods. Moreover, we conducted an investigation into the various components of constructing argumented control flow graphs for graph neural networks (GNNs) and evaluated their relative importance. Overall, our findings demonstrate the effectiveness of our proposed approach and provide insights into the construction of argumented control flow graphs for GNNs. Further research in this area is expected to lead to even more significant improvements in binary analysis using deep learning.

Chapter 4

Conclusions

In this paper, we have explored two different approaches for resolving indirect control flow in programs. In the first part of the paper, we debloat the shared library without resolving each indirect control flow. Instead, our observation is solving a set of indirect control flow is enough for debloating purpose. Our results showed our method achieved similar debloating performance with static linking and with practice correctness.

In the second part of the paper, we explored the use of graph neural networks (GNNs) for resolving indirect calls. We presented a novel approach that represent reference as edges in a augmented control flow graph to predict the targets of indirect calls. Our evaluation showed that our approach achieves high accuracy and outperforms existing techniques.

Our work highlights the importance of resolving indirect control flow in programs for various purposes and demonstrates that different techniques can be applied depending on the specific context and requirements of the application. Our results also suggest that machine learning techniques, such as GNNs, can be effectively used for resolving indirect control flow in programs. However, there are still many open research challenges in this field. We hope that our work will motivate further research in this area.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information and System Security*, 13(1), November 2009.

[2] Ali Abbasi, Jos Wetzels, Thorsten Holz, and Sandro Etalle. Challenges in Designing Exploit Mitigations for Deeply Embedded Systems. In *Proceedings of the 4th IEEE European Symposium on Security and Privacy (EuroS&P'19)*, 2019.

[3] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. Nibbler: Debloating Binary Shared Libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC'19)*, 2019.

[4] Onur ALANBEL. Developing MIPS Exploits to Hack Routers. BGA Information Security Whitepaper, April 2015.

[5] Naif Saleh Almakhdhub, Abraham A. Clements, Saurabh Bagchi, and Mathias Payer. μRAI: Securing Embedded Systems with Return Address Integrity. In *Proceedings of the 2020 Network and Distributed System Security Symposium (NDSS'20)*, 2020.

[6] Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, Herbert Bos, and Michael Franz. BinRec: Dynamic Binary Lifting and Recompilation. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys'20)*, 2020.

[7] Erik Andersen. uClibc is a small C standard library intended for Linux kernel-based OS on embedded systems and mobile devices. https://www.uclibc.org/, 2022.

[8] AspenCore. 2019 Embedded Markets Study. http://tiny.cc/a4mwtz, November 2019.

[9] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. Less is More: Quantifying the Security Benefits of Debloating Web Applications. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security'19)*, 2019.

[10] Gogul Balakrishnan and Thomas Reps. WYSINWYX: What You See is Not What You eXecute. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(6), August 2010.

[11] Sebastian Banescu and Alexander Pretschner. Chapter Five - A Tutorial on Software Obfuscation. Advances in Computers. Elsevier, 2018.

[12] Markus Bauer, Ilya Grishchenko, and Christian Rossow. Typro: Forward cfi for c-style indirect function calls using type propagation. In *Proceedings of the 38th Annual Computer Security Applications Conference*. Association for Computing Machinery, 2022.

[13] Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural computation*, 15(6):1373–1396, 2003.

[14] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the 2005 USENIX Annual Technical Conference (ATC'05)*, 2005.

[15] Ketan Bhardwaj, Matt Saunders, Nikita Juneja, and Ada Gavrilovska. Serving Mobile Apps: A Slice at a Time. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys'19)*, 2019.

[16] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS'11)*, 2011.

[17] Bobby Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. JShrink: In-Depth Investigation into Debloating Modern Java Applications. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)*, 2020.

[18] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-Flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys*, 50(1), April 2017.

[19] Chen Cao, Le Guan, Jiang Ming, and Peng Liu. Device-agnostic Firmware Execution is Possible: A Concolic Execution Approach for Peripheral Emulation. In *Proceedings of the 36th Annual Computer Security Applications Conference (ACSAC'20)*, 2020.

[20] Nicholas Carlini and David Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In *Proceedings of the 23th USENIX Conference on Security Symposium (USENIX Security'14)*, 2014.

[21] Tim Carrington. Remote Code Execution (CVE-2018-5767) Walkthrough on Tenda AC15 Router. https://fidusinfosec.com/remote-code-execution-cve-2018-5767/, February 2018.

[22] Ines Chami, Adva Wolf, Da-Cheng Juan, Frederic Sala, Sujith Ravi, and Christopher Ré. Low-dimensional hyperbolic knowledge graph embeddings. pages 6901–6914, July 2020. doi: 10.18653/v1/2020.acl-main.617. URL https://aclanthology.org/2020.acl-main.617.

[23] Daming Dominic Chen, Manuel Egele, Maverick Woo, and David Brumley. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS'16)*, 2016.

[24] Jake Christensen, Ionut Mugurel Anghel, Rob Taglang, Mihai Chiroiu, and Radu Sion. DECAF: Automatic, Adaptive De-bloating and Hardening of COTS Firmware. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security'20)*, 2020.

[25] Abraham A. Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security'20)*, 2020.

[26] Lucian Cojocar, Taddeus Kroes, and Herbert Bos. JTR: A Binary Solution for Switch-Case Recovery. In *Proceedings of the 2017 International Symposium on Engineering Secure Software and Systems*, 2017.

[27] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*, chapter 4.4, pages 258–276. Addison-Wesley Professional, 2009.

[28] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A Large-Scale Analysis of the Security of Embedded Firmwares. In *Proceedings of the 23rd USENIX Conference on Security Symposium (USENIX Security'14)*, 2014.

[29] Stephen Crane, Per Larsen, Stefan Brunthaler, and Michael Franz. Booby Trapping Software. In *Proceedings of the 2013 New Security Paradigms Workshop (NSPW'13)*, 2013.

[30] Yaniv David, Nimrod Partush, and Eran Yahav. FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*, 2018.

[31] Yaniv David, Uri Alon, and Eran Yahav. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.

[32] Austin Derrow-Pinion, Jennifer She, David Wong, Oliver Lange, Todd Hester, Luis Perez, Marc Nunkesser, Seongjae Lee, Xueying Guo, Brett Wiltshire, et al. Eta prediction with graph neural networks in google maps. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, pages 3767–3776, 2021.

[33] CVE Details. Security Vulnerabilities (Memory Corruption). https://www.cvedetails.com/vulnerability-list/opmemc-1/memory-corruption.html, 2022.

[34] Alessandro Di Federico and Giovanni Agosta. A Jump-Target Identification Method for Multi-Architecture Static Binary Translation. In *Proceedings of the 2016 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2016.

[35] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 472–489. IEEE, 2019.

[36] Gregory J. Duck, Xiang Gao, and Abhik Roychoudhury. Binary Rewriting without Control Flow Recovery. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'20)*, 2020.

[37] Vijay Prakash Dwivedi and Xavier Bresson. A generalization of transformer networks to graphs. *arXiv preprint arXiv:2012.09699*, 2020.

[38] Vijay Prakash Dwivedi, Chaitanya K Joshi, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. Benchmarking graph neural networks. *arXiv preprint arXiv:2003.00982*, 2020.

[39] Eta Labs. Comparison of C/POSIX Standard Library Implementations for Linux. http://www.etalabs.net/compare_libcs.html, 2022.

[40] Rod Evans. Static Linking - where did it go? https://blogs.oracle.com/solaris/post/static-linking-where-did-it-go, December 2004.

[41] Bo Feng, Alejandro Mera, and Long Lu. P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security'20)*, 2020.

[42] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable Graph-based Bug Search for Firmware Images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*, 2016.

[43] The Apache Software Foundation. Apache HTTP Test Project. https://httpd.apache.org/test/, 2018.

[44] GCC Manual. Options for Code Generation Conventions. https://gcc.gnu.org/onlinedocs/gcc/Code-Gen-Options.html, 2022.

[45] GCC Manual. MIPS Options. https://gcc.gnu.org/onlinedocs/gcc/MIPS-Options.html, 2022.

[46] Masoud Ghaffarinia and Kevin W. Hamlen. Binary Control-Flow Trimming. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS'19)*, 2019.

[47] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. Temporal System Call Specialization for Attack Surface Reduction. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security'20)*, 2020.

[48] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International conference on machine learning*, pages 1263–1272. PMLR, 2017.

[49] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out Of Control: Overcoming Control-Flow Integrity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 2014.

[50] GrammaTech. Office of Naval Research awards GrammaTech $9M for Cyber-Hardening Security Research. https://news.grammatech.com/onr-awards-grammatech-9m-for-cyber-hardening-research, October 2017.

[51] Aaron Guzman and Aditya Gupta. *IoT Penetration Testing Cookbook: Identify Vulnerabilities and Secure Your Smart Devices*. Packt Publishing, November 2017.

[52] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

[53] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1667–1680, 2018.

[54] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*, 2018.

[55] Hex-Rays. The IDA Pro Disassembler and Debugger. https://www.hexrays.com/products/ida/, 2021.

[56] Patrick Horgan. Linux Program Start Up—How the heck do we get to main()? http://dbp-consulting.com/tutorials/debugging/linuxProgramStartup.html, 2022.

[57] Zecong Hu. GitHub Cloner and Compiler. https://github.com/huzecong/ghcc, 2021.

[58] Fortune Business Insights. Internet of Things (IoT) Market Analysis. http://tiny.cc/lsj1tz, July 2019.

[59] Muhui Jiang, Yajin Zhou, Xiapu Luo, Ruoyu Wang, Yang Liu, and Kui Ren. An Empirical Study on ARM Disassembly Tools. In *Proceedings of the 29th International Symposium on Software Testing and Analysis (ISSTA'20)*, 2020.

[60] Yufei Jiang, Dinghao Wu, and Peng Liu. JRed: Program Customization and Bloatware Mitigation Based on Static Analysis. In *Proceedings of the 40th IEEE Annual Computer Software and Applications Conference (COMPSAC'16)*, 2016.

[61] Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings. 2022.

[62] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. 1:2, 2019.

[63] Sun Hyoung Kim, Cong Sun, Dongrui Zeng, and Gang Tan. Refining indirect call targets at the binary level. In *NDSS*, 2021.

[64] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*, 2014.

[65] ReFirm Labs. Binwalk: Firmware Analysis Tool. https://github.com/ReFirmLabs/binwalk, 2022.

[66] William Landi. Undecidability of Static Analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[67] William Landi. Recovery of Jump Table Case Statements from Binary Code. *Science of Computer Programming*, 40, February 2001.

[68] Yongjun Lee, Hyun Kwon, Sang-Hoon Choi, Seung-Ho Lim, Sung Hoon Baek, and Ki-Woong Park. Instruction2vec: Efficient preprocessor of assembly code to detect software weakness with cnn. *Applied Sciences*, 9(19):4086, 2019.

[69] Xuezixiang Li, Yu Qu, and Heng Yin. Palmtree: learning an assembly language model for instruction embedding. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3236–3251, 2021.

[70] Jian Lin, Liehui Jiang, Yisen Wang, and Weiyu Dong. A Value Set Analysis Refinement Approach Based on Conditional Merging and Lazy Constraint Solving. *IEEE Access*, 7, 2019.

[71] LLVM Project. Architecture & Platform Information for Compiler Writers. https://llvm.org/docs/CompilerWriterInfo.html, 2022.

[72] Kangjie Lu and Hong Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1867–1881, 2019.

[73] Andrew L. Maas. Rectifier nonlinearities improve neural network acoustic models. 2013.

[74] Xiaozhu Meng and Weijie Liu. Incremental CFG Patching for Binary Rewriting. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*, 2021.

[75] Xiaozhu Meng and Barton P. Miller. Binary Code is Not Easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16)*, 2016.

[76] Melissa Michael. Attack Landscape H1 2019: IoT, SMB traffic abound. http://tiny.cc/jsj1tz, September 2019.

[77] Federico Monti, Fabrizio Frasca, Davide Eynard, Damon Mannion, and Michael Bronstein. Fake news detection on social media using geometric deep learning. *ICLR*, 2019.

[78] Gianluca Pacchiella. CVE-2020-8423: Exploiting the TP-LINK TL-WR841N V10 Router. https://ktln2.org/2020/03/29/exploiting-mips-router/, March 2020.

[79] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 833–851. IEEE, 2021.

[80] Alejandro Parodi. Exploiting Routers: Just Another TP-Link 0-Day. https://www.secsignal.org/en/news/exploiting-routers-just-another-tp-link-0day/, November 2018.

[81] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. Trex: Learning execution semantics from micro-traces for binary similarity. *arXiv preprint arXiv:2012.08680*, 2020.

[82] Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana. Stateformer: fine-grained type recovery from binaries using generative state modeling. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 690–702, 2021.

[83] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. BlankIt Library Debloating: Getting What You Want Instead of Cutting What You Don't. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'20)*, 2020.

[84] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. RAZOR: A Framework for Post-Deployment Software Debloating. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security'19)*, 2019.

[85] Chenxiong Qian, HyungJoon Koo, ChangSeok Oh, Taesoo Kim, and Wenke Lee. Slimium: Debloating the Chromium Browser with Feature Subsetting. In *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS'20)*, 2020.

[86] Anh Quach, Aravind Prakash, and Lok Yan. Debloating Software through Piece-Wise Compilation and Loading. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)*, 2018.

[87] G. Ramalingam. The Undecidability of Aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, September 1994.

[88] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. Cimplifier: Automatically Debloating Containers. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*, 2017.

[89] Red Hat Customer Portal. Static Linking Not Supported in Red Hat Enterprise Linux 8. https://access.redhat.com/articles/rhel8-abi-compatibility, May 2019.

[90] Nilo Redini, Ruoyu Wang, Aravind Machiry, Yan Shoshitaishvili, Giovanni Vigna, and Christopher Kruegel. BinTrimmer: Towards Static Binary Debloating Through Abstract Interpretation. In *Proceedings of the 16th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'19)*, 2019.

[91] Xiaolei Ren, Michael Ho, Jiang Ming, Yue Lei, and Li Li. Unleashing the Hidden Power of Compiler Optimization on Binary Code Difference: An Empirical Study. In *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'21)*, 2021.

[92] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Transactions on Information and System Security*, 15(1), March 2012.

[93] Jonathan Salwan. ROPgadget - Gadgets finder and auto-roper. http://shell-storm.org/project/ROPgadget, 2011.

[94] Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter Battaglia. Learning to simulate complex physics with graph networks. In *International Conference on Machine Learning*, pages 8459–8468. PMLR, 2020.

[95] Jason Sattler. Attack Landscape H2 2019: An Unprecedented Year for Cyber Attacks. http://tiny.cc/esj1tz, March 2020.

[96] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

[97] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *The Semantic Web: 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3–7, 2018, Proceedings 15*, pages 593–607. Springer, 2018.

[98] Tara Seals. Critical Cisco Bug in VPN Routers Allows Remote Takeover. https://threatpost.com/critical-cisco-bug-vpn-routers/168449/, August 2021.

[99] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86). In *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS'07)*, 2007.

[100] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. TRIMMER: Application Specialization for Code Debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)*, 2018.

[101] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P'16)*, 2016.

[102] Standard Performance Evaluation Corporation. Building the SPEC CPU2017 Toolset. https://www.spec.org/cpu2017/Docs/tools-build.html, 2017.

[103] Yulei Sui and Jingling Xue. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction*, 2016.

[104] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P'13)*, 2013.

[105] The Santa Cruz Operation. System V Application Binary Interface MIPS RISC Processor Supplement, 3rd Edition. https://refspecs.linuxfoundation.org/elf/mipsabi.pdf, February 1996.

[106] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P'16)*, 2016.

[107] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.

[108] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making Reassembly Great Again. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS'17)*, 2017.

[109] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *NDSS*, 2017.

[110] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable Disassembling. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security'15)*, 2015.

[111] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable disassembling. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 627–642, 2015.

[112] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*, 2016.

[113] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. Egalito: Layout-Agnostic Binary Recompilation. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, 2020.

[114] Jianliang Wu, Ruoyu Wu Daniele Antonioli, Mathias Payer, Nils Ole Tippenhauer, Dongyan Xu, Dave (Jing) Tian, and Antonio Bianchi. LIGHTBLUE: Automatic Profile-Aware Debloating of Bluetooth Stacks. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security'21)*, 2021.

[115] Karim Yaghmour, Jon Masters, Gilad Ben-Yossef, and Philippe Gerum. *Building Embedded Linux Systems: Concepts, Techniques, Tricks, and Traps*. O'Reilly Media, second edition, August 2008.

[116] Lyon Yang. Exploiting Buffer Overflows on MIPS Architectures. Hack In The Box Security Conference 2015 Whitepaper, October 2015.

[117] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *Proceedings of the 2014 Network and Distributed System Security Symposium (NDSS'14)*, 2014.

[118] W. Zhu, Z. Feng, Z. Zhang, J. Chen, Z. Ou, M. Yang, and C. Zhang. Callee: Recovering call graphs for binaries with transfer and contrastive learning. In *2023 2023 IEEE Symposium on Security and Privacy (SP) (SP)*, pages 1953–1970, Los Alamitos, CA, USA, may 2023. IEEE Computer Society. doi: 10.1109/SP46215.2023.00112. URL https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.00112.

[119] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. Neural machine translation inspired binary code similarity comparison beyond function pairs. *arXiv preprint arXiv:1808.04706*, 2018.