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

Haotian Zhang
University of Texas at Arlington
Arlington, Texas, USA

Mengfei Ren
University of Texas at Arlington
Arlington, Texas, USA

Yu Lei
University of Texas at Arlington
Arlington, Texas, USA

Jiang Ming
University of Texas at Arlington
Arlington, Texas, USA

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.

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02.
https://doi.org/10.1145/3503222.3507768

CCS CONCEPTS

• **Security and privacy → Embedded systems security**.

KEYWORDS

software debloating, static analysis, embedded systems

ACM Reference Format:
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), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3503222.3507768

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 [46]. With the IoT boom taking place, cyberattacks on embedded devices, which surged 300% in 2019 [57, 71], 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 [25, 27, 34]. Although memory corruption vulnerabilities [78] have been around for decades, they also dominate the share of top-rated threats in embedded devices [28]. 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 [69].

Firmware developers rely on C and C++ shared libraries (e.g., uClibc [7]) for fast prototyping and development [85]. 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 [63]. 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 [69]. 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 [17, 20, 22, 33, 87]. 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 [3, 9, 13, 15, 21, 38, 39, 43, 48, 60–63, 65, 67, 74, 84]. In particular, library debloating techniques [3, 60, 63] 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 [82] and control-flow integrity schemes [16, 49].

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 [63] and BlankIt [60] 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* [27].

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, 47, 56, 68] 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) [79] 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 [36, 54].

We develop an input-profiling-agnostic, static debloating system for MIPS stripped binaries, named *μTrimmer*, to eliminate unused code present in shared libraries. Note that the effects of multi-entry functions and tail call optimization [56] 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 [51, 64]. 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 [26] and delivers thinned libraries.

We build μTrimmer on top of angr [75] 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 caused by static linking is taken by recent work [3, 63] as an upper bound of library debloating; μTrimmer's debloating capability competes with the static linking results and outperforms piece-wise [63] 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 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.

2.1 Code-Reuse Attacks on IoT Devices

The proliferation of the IoT market makes embedded devices a lucrative target for cybercriminals. 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

Table 1: Comparison of representative library debloating approaches.

	Piece-wise [63]	Nibbler [3]	BlankIt [60]	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

network [19, 42, 58, 59, 72]. Among various attacks against the IoT ecosystem, code-reuse attacks [69] 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 [73].

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 [31]. Even so, uClibc is still a fertile land to search ROP gadgets [4, 86]. 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 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 [3, 9, 13, 15, 21, 38, 39, 43, 48, 60–63, 65, 67, 74, 84]. The debloating targets range from program binaries [38, 61, 67], Java applications [15, 48], Docker containers [65], mobile/web applications [9, 13, 62], UEFI firmware [21], Bluetooth stacks [84], and shared libraries [3, 60, 63]. Most existing works assume the availability of source code, sample inputs as the usage profile, or runtime support. For example, DECAF [21] 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 [3, 60, 63]. We compare them with our work in Table 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 [63] 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 [77] 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 (~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 [82] and control-flow integrity defenses [16]. 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 [60] 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.

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 [47]. In contrast, MIPS binaries do not have such complicated properties, making reliably disassembling MIPS binaries a solved problem.

Furthermore, MIPS Application Binary Interface (ABI) [79] 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) [37]. 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.

2.4 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. [67] 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 [53]. 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.

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 1 illustrates the architecture of μTrimmer. ❶ ~ ❹ represent the following four workflow steps.

1. Preprocess Given a firmware image, we adopt Binwalk [50] 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 [56] 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.

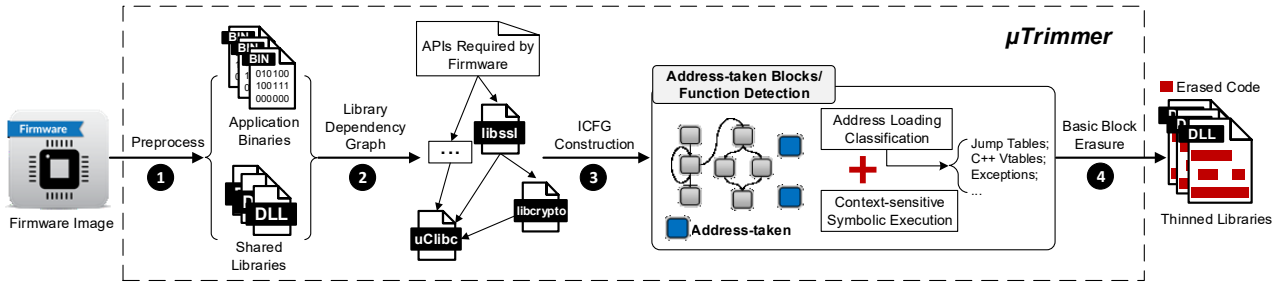

Figure 1: The overview of the μTrimmer. The whole process consists of four steps.

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 [6, 30, 55, 80, 81, 83] 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 interprocedural control flow graph (ICFG) construction.

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`) [45]; these functions

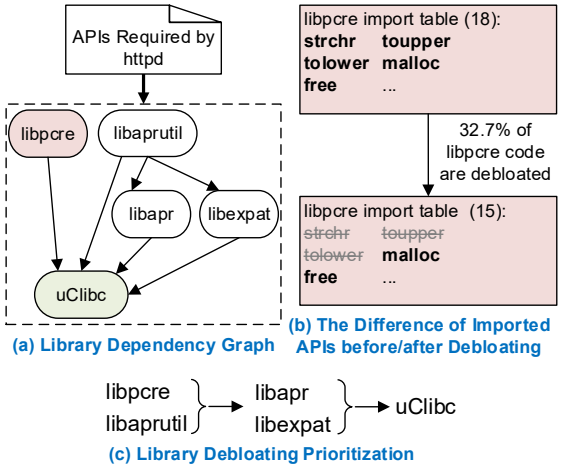

Figure 2: Library dependency graph of httpd (Apache).

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(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(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 successor nodes as the input of ICFG construction.

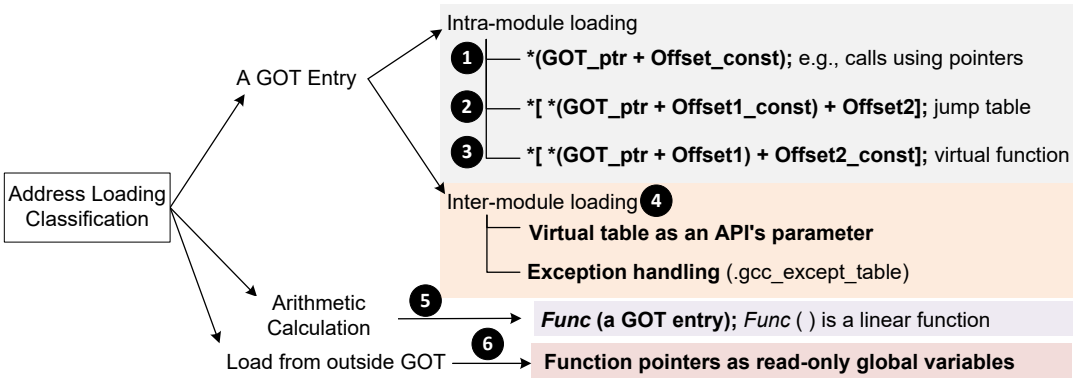

Figure 3: Six address loading patterns according to how indirect control flow targets are loaded from memory.

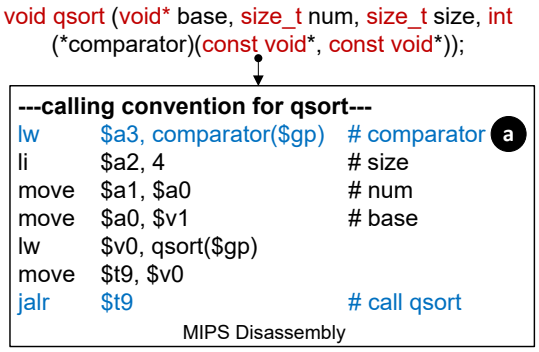

Figure 4: An example of callback function.

Figure 2(c) shows the prioritization of library debloating, which follows the topological ordering of Figure 2(a).

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) [56] 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 [1, 3] lack a complete picture of AT function types, hindering their effectiveness. We present a new, comprehensive taxonomy that covers all types of indirect jumps/calls.

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 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 3). Function calls using pointers (e.g., callback functions) also belong to this type. As shown in Figure 4, when a variable is assigned as a function pointer, the compiler generates instructions to obtain its address via GOT-relative addressing (ⓐ in Figure 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 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 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 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 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

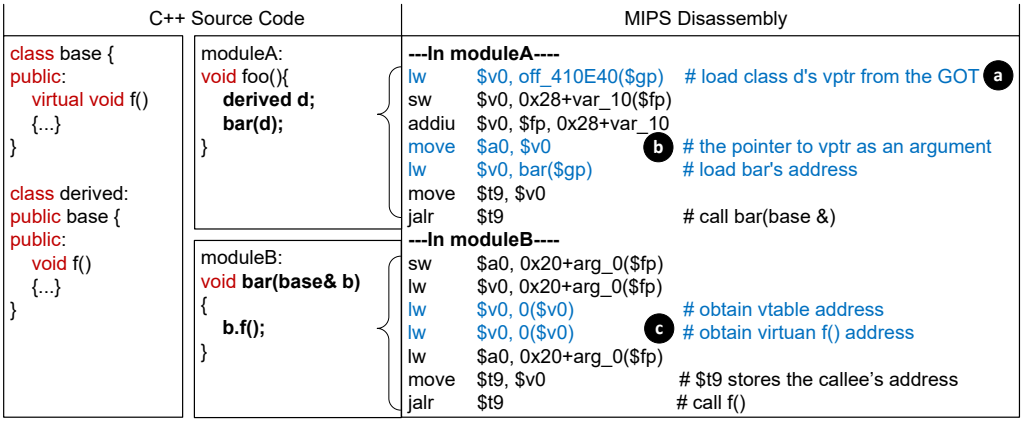

Figure 5: An example of inter-module address loading from libstdc++. We simplify the code snippet for the easy presentation purpose. The moduleA only loads a virtual table's address and passes it as an API's argument. Only the moduleB loads the virtual function's address from the moduleA's virtual table (within the scope of moduleA's GOT).

Table 2: The distribution of address loading types (see Figure 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%

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 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.

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.

5.2.1 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 [23, 29, 52], 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.

5.2.2 Virtual Functions. In Figure 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 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.

5.2.3 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 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
```

5.2.4 Arithmetic Calculation. In Table 2, 10.5% of uClibc's control flow target addresses are calculated from a GOT entry. Listing 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.

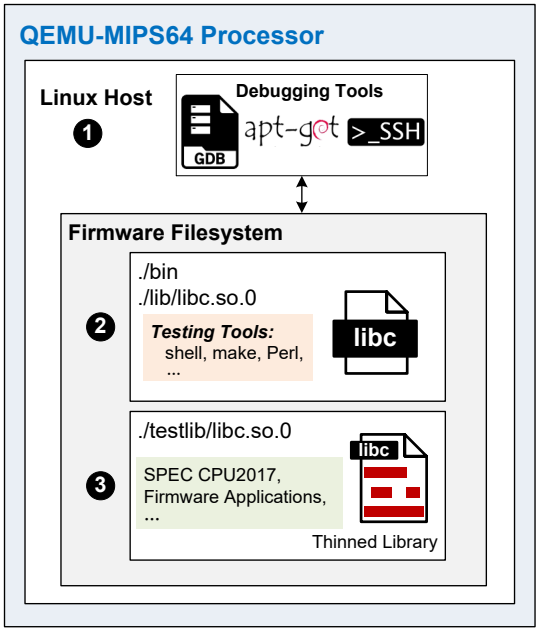

Figure 6: Testing environment setup.

6 EVALUATION

We developed μTrimmer on top of a binary code analysis platform, angr [75]. 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.

Environment Setup As shown in Figure 6, we build our testing environment using QEMU [12] 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 6), which are necessary to efficiently develop μTrimmer. All

Table 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

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 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 [3, 63].

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 [76]. 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.

6.1 Code Reduction and Correctness

Code Reduction Metrics Table 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 4.

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 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 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 [35]. 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 [67] to resolve possible targets of an indirect control flow.

Table 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%

Table 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%

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.

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 [32], and Red Hat Enterprise Linux 8 does not support static linking anymore [66]. 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 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 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 3), while static linking can correctly remove functions in the ".data.rel.ro" section.

Directly comparing related library debloating work [3, 60, 63] is infeasible because of their specific assumptions (e.g., source code and runtime support) and different platform requirements. Fortunately, both piece-wise [63] 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.

Table 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%

Table 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

6.3 Gadget Reduction

We use ROPgadget [70] to measure three common gadget types: syscall [73], stack pointer update (SPU) [40], and jump-oriented programming (JOP) [14]. Note that MIPS does not contain return instructions and instructions like "call *(memory)," thus conventional gadget types, such as ret-based gadgets [73] and call-oriented programming (COP) gadgets [18], are not available in MIPS. As shown in Table 6, μTrimmer seems to be very effective in removing the syscall gadget class, whose reduction ratio is much higher than the respective 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.

6.4 Firmware Image Debloating

Piece-wise [63] 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 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 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 [56]. We use IDA Pro's function recovery heuristics [44] 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 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.

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 [47] 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 [11, 24] 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. Piece-wise [63] 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 [6, 30, 55, 80, 81, 83]. Therefore, our future work is to explore binary rewriting to achieve the goal of code size reduction.

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 [41].

ACKNOWLEDGMENTS

We sincerely thank ASPLOS 2022 anonymous reviewers for their insightful comments and Dr. Santosh Nagarakatte for helping us improve the paper throughout the shepherding process. This work was supported by the National Science Foundation (NSF) grants (CNS-1850434 and CNS-2128703) and Cisco Research Grant ("RFP-21-07").

A ARTIFACT APPENDIX

A.1 Artifact Check-List (Meta-Information)

- **Program:** A SPEC CPU 2017 benchmark suite (version 1.0.2) is needed for this Artifact Evaluation (compilation script is included). Other testing programs (busybox, nano, apache, etc) used in our experiments are included.
- **Compilation:** A MIPS compiler is needed.A MIPS uclibc gcc 6.3.0 toolchain is included.
- **Transformations:** Binary lifting to IR with angr.
- **Run-time environment:** The artifact is an ovf format virtual machine, exported from VMware workstation 14 Pro. Inside the image is a Ubuntu 16.04 with root access to both terminal and GUI. Python 3.6 is also required.
- **Hardware:** A x86-64 machine with at least 16GB memory for virtual machine execution.
- **Execution:** μTrimmer itself processes a library at a time. For each shared library used by a program, it may cost 15 minutes to 4 hours for execution. Note: each program may contain multiple libraries.
- **Metrics:** We use code reduction size and ratio of each library as our debloating efficiency. We further report how many functions are full or partial removed as informational data.
- **Output:** The output consists of a debloated library and a summary text file (which is a copy of the console output). The code reduction size and the ratio is in the output text file. We also report how many functions are fully or partially debloated, which are collected with IDA Pro and provided Python scripts.
- **Experiments:** For each experiment, we have provided related scripts and manual steps.
- **How much disk space required (approximately)?:** 80GB.
- **How much time is needed to prepare workflow (approximately)?:** 4 hours.
- **How much time is needed to complete experiments (approximately)?:** A couple of days to run all the experiments.
- **Archived (provide DOI)?:** 10.5281/zenodo.5746505

A.2 Description

A.2.1 How to access. Download the VM image from the zenodo at https://zenodo.org/record/5746505 and use the following credential information to login.

```
VM info:
Default hardware setting:
Memory: 16GB
Processor: 8

System: ubuntu-16.04.5
Username: uTrimmer
Pw: mu.trimmer
QEMU: Debian
Username: root
Pw: mu.trimmer
```

A.2.2 Software dependencies. Running μTrimmer itself does not need additional software. However, to evaluate the result, IDA Pro 7.1, QEMU 5.1, binwalk 2.2 is required. We do not include IDA Pro in the VM image due to it is a commercial software. But we include the output of testcases from IDA Pro, in case someone needs to evaluate the result without IDA Pro. QEMU and binwalk are already installed in the VM image.

A.2.3 Data sets. SPEC CPU 2017 (v1.0.2) and Apache, busybox, prel, openssl, nano, unzip are used for evaluation. We do not include SPEC CPU 2017 in the VM image due to it is a commercial software. But the compilation toolchain and compilation configuration file are provided. Alternatively, pre-compiled Apache, busybox, prel, openssl, nano, unzip binaries are provided in the VM image.

A.3 Installation

We provide a VM image which contains all the necessary packages in zenodo, it is recommand to directly using that image. The VM includes three files, Ubuntu 16.04.05.mf, Ubuntun 16.04.05.ovf, and Ubuntu 16.04.05.vmdk.

To start with a fresh Ubuntu 16.04, first install python 3.6 and basic packages and create a virtual environment.

Download the source.zip from zenodo, extract and execute install.sh after previous setup.

A.4 Evaluation and Expected Results

A.4.1 SPEC CPU2017 Setup. Since SPEC CPU 2017 is a commercial software, we do not install it in the VM image. Instead, we provide the following instruction for user to compile SPEC if it is available. The SPEC CPU 2017 should be compiled with a MIPS compile toolchain that is provided at '~/toolchaincxx', and a sample configuration file that is named as cpu2017_linux_mips.cfg.

After extract the SPEC CPU 2017, copy the configuration file into SPEC CPU folder and compile. After the compilation, copy the generated binary into ~/uTrimmer_test/data/cpu2017 folder and rename it to the specific name instructed by the README file inside the folder.

A.4.2 Reproducing Result in Table 3. Table 3 summarizes the code reduction metrics achieved by μTrimmer. The "Table3::Library Statistic" column is some statistic description of input, which is manually collected for each library with readelf and IDA. The "Table3::Code Reduction Metrics" column is μTrimmer debloating results.

Step 1: For each program of SPEC and firmware applications, we create a separate Python script to execute it with μTrimmer, such as 502.py and apache.py. To get the the value in "Table 3::Code Reduction Metrics::Size" and "Table 3::Code Reduction Metrics::Ratio", user needs to execute the scripts that are saved in folder ~/uTrimmer_test/final_tests/.

After the execution, a summary of output is saved in ~/uTrimmer_test/output (*.result) and the debloated shared library is also located in the same folder. Open the *.result with text editor. There are three key values at the end of the file.

For a program has multiple library dependencies, you need manually sum up each library result with the following formula.

$$Ratio = sum\_deleted\_size/sum\_.text\_size$$

The results of "Code Reduction Metrics::#Full", "Code Reduction Metrics::#Partial" and "Code Reduction Metrics::Partial Size" are manually collected with IDA Pro for each program under test.

Step 2: *If IDA Pro is not available, please skip this step, we already provide a list of result files from IDA Pro in ~/uTrimmer_test/output/func/.

When IDA Pro is available, user needs to manually disassemble the original shared library of each program in IDA Pro. Original shared libraries are located in ~/uTrimmer_test/data/libs. After the disassemble finished, run our IDA scripts by choosing in IDA "File" - "Script file ..." select ~/uTrimmer_test/ida_func.py. A function list will be saved as a txt file with the library's name under the folder ~/uTrimmer_test/output/func after executing the script file.

Step 3: To compare the μTrimmer result file with IDA function result, edit '~/uTrimmer_test/compare_ida_func.py', and modify the first 'progname' varible with the name of the test script that are saved in folder ~/uTrimmer_test/final_tests/, then run the compare_ida_func.py script.

At the end of the console output, the following result is displayed,

- **fullremoved num**: corresponds to "Table 3::Code Reduction Metrics::#Full"
- **partial num**: corresponds to "Table 3::Code Reduction Metrics::#Partial"
- **partial_size**: corresponds to "Table 3::Code Reduction Metrics::PartialSize"

Step 4(Optional): To verify the correctness of debloating results, we prepared a MIPS image and QEMU for firmware emulation. To evaluate the correctness in QEMU, user needs to copy both the test program and debloated library to QEMU environment. Then switch to the testing environment by using chroot in QEMU. The test program should execute successfully in QEMU after debloating by our framework μTrimmer.

Additionally, all programs under test are compiled with option '-Wl,-rpath=/testlib'. The test program will load the shared library at /testlib first rather than the original library. To verify this library dependency, run readelf with the test program in Ubuntu. The output should shows that variable "Library rpath" points to "/testlib".

A.4.3 Reproducing Result in Table 5. We compared μTrimmer's debloating result with the static linker with garbage collection enabled on Apache web server. The result is shown in Table 5. To get the debloating result from μTrimmer, run httpd_static.py script like in Section A.4.2.

The .text size of output represents the value in "Table 5::Dynamic" column of the dynamic library.

The values in "Table 5::μTrimmer::%Redu." are the same as "Table 3::Code Reduction Metrics::Ratio". However, the value in "Table 5::μTrimmer::Size" is the remaining Size rather than the removing Size shown in Table 3, which is calculated by the following formula,

$$Size = .text\_size - deleted\_size$$

"Table 5::Static" column presents the static linker preserved code size during linking process. Run parse_httpd_static.py script to get the "Table 5::Static" result. The output displays the size preserved by static linker, which corresponds to "Table 5::Static::Size".

The reduction result of static linker is manually calculated by the following formula,

$$Static :: \%Redu. = 1 - (Dyn. :: Size - Static :: Size)/(Dyn. :: Size)$$

A.4.4 Reproducing Result in Table 6. We measured three common ROP gadget types with μTrimmer on SPEC CPU2017 and firmware applications, which is shown in Table 6. Once the debloating process is done, which is performed in Section A.4.2, run ROP_calc.py script.

The code reduction ratio result of each program under test is collected from Table 3. The console output of ROP_calc.py should show the gadget reduction ratio of each type in Table 6. The execution result should show that μTrimmer is effective in removing the syscall gadget class, whose reduction ratio is much higher than the respective code reduction.

A.4.5 Reproducing Result in Table 7. We conduct an experiment on μTrimmer with a real-world embedded device's firmware, which is a wireless router Archer A10 V1_190305 from TP-Link. The firmware version is v1 and can be downloaded at TP-Link's official website at https://www.tp-link.com/us/support/download/archer-a10/v1/#Firmware. We already provide the extracted libraries from the firmware which locates in ~/uTrimmer_test/data/TPlink_a10/lib. To extract the firmware, run binwalk by replacing the binary file with the download firmware,

The library statistic are manually collected with IDA Pro, which are saved in ~/uTrimmer_test/data/TPlink_a10.

To calculate the code reduction size and ratio of each library, run tplink_a10.py script. The output is saved in ~/uTrimmer_test/output, similar with Section A.4.2. The number of functions of fully or partially removed in each library is collected with IDA Pro by performing the same steps shown in Section A.4.2.

A.5 Experiment Customization

Using the provided toolchain, one can compile other programs and libraries for testing. Toolchain is located at '~/toolchaincxx', it support both C/C++ programs.

After the compilation is done, the shared libraries should be put into the '~/uTrimmer_test/data/libs' folder. Then rewrite a testing script inside the '~/uTrimmer_test/final_test' folder, replacing the program name and path, and the shared library names.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information and System Security* 13, 1, Article 4 (November 2009), 40 pages.

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

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

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

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

[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. 2020. BinRec: Dynamic Binary Lifting and Recompilation. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys'20).*

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

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

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

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

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

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

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

[14] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. 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)*.

[15] Bobby Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. 2020. 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)*.

[16] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-Flow Integrity: Precision, Security, and Performance. *Comput. Surveys* 50, 1, Article 16 (April 2017), 33 pages.

[17] Chen Cao, Le Guan, Jiang Ming, and Peng Liu. 2020. 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)*.

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

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

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

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

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

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

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

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

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

[27] Yaniv David, Nimrod Partush, and Eran Yahav. 2018. 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)*.

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

[29] Alessandro Di Federico and Giovanni Agosta. 2016. 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*.

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

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

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

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

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

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

[36] GCC Manual. [online]. MIPS Options. https://gcc.gnu.org/onlinedocs/gcc/MIPS-Options.html.

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

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

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

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

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

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

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

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

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

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

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

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

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

[50] ReFirm Labs. [online]. Binwalk: Firmware Analysis Tool. https://github.com/ReFirmLabs/binwalk.

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

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

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

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

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

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

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

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

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

[60] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. 2020. 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)*.

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

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

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

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

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

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

[67] Nilo Redini, Ruoyu Wang, Aravind Machiry, Yan Shoshitaishvili, Giovanni Vigna, and Christopher Kruegel. 2019. 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)*.

[68] Xiaolei Ren, Michael Ho, Jiang Ming, Yue Lei, and Li Li. 2021. 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)*.

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

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

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

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

[73] Hovav Shacham. 2007. 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)*.

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

[75] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. 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)*.

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

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

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

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

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

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

[82] 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. 2016. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*.

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

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

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

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

[87] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. 2014. 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)*.