PLDI: G: Smaller and More Secure: Static Debloating of MIPS Firmware Shared Libraries

Haotian Zhang (Advisor: Jiang Ming), haotian.zhang@mavs.uta.edu
University of Texas at Arlington, Arlington, TX, 76019

1 PROBLEM & MOTIVATION

Over the past few years, the spotlight has been on the Internet of Things (IoT) market due to the sheer amount being deployed worldwide. With the IoT boom taking place, cyberattacks on embedded devices are now accelerating at an unprecedented rate [1]. 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 [2]. Although memory corruption vulnerabilities [3] have been around for decades, they also dominate the share of top-rated threats in embedded devices [4]. For example, buffer overflow bugs in WiFi routers and smart home devices allows remote attackers to completely take over the device and enter the home network [5]. 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 [6].

Firmware developers rely on C/C++ shared libraries for fast prototyping and development (e.g., uClibc [7]). 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 [8]. 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 [6]. 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 [9–11]. Therefore, these limitations restrict the adoption of expensive ROP countermeasures to secure embedded systems.

2 BACKGROUND & RELATED WORK

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. In particular, library debloating techniques [8, 12, 13] 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 [14] and control-flow integrity schemes [15].

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.

2.1 Shared Library Debloating

In the literature, several techniques have been recently proposed to detect and remove unused code from shared libraries [8, 12, 13]. We compare them with our work, named *µTrimmer*, 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 [8] 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 [16] 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 [12] 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 [14] and control-flow integrity defenses [15]. 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 [13] 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

Table 1: Comparison of representative library debloating approaches.

	Piece-wise [8]	Nibbler [12]	BlankIt [13]	μTrimmer (This Work)
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

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.

2.2 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 (CFGs). 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 [17]. In contrast, MIPS binaries do not have such complicated properties, making reliably disassembling MIPS binaries a solved problem.

Furthermore, MIPS Application Binary Interface (ABI) [18] 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) [19]. 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.

2.3 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 [20] 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. [21] 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 [22]. 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/static object). Therefore, they are possible targets of indirect jump/call instructions. Control flow integrity [23] 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 [12] 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 UNIQUENESS OF THE APPROACH

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

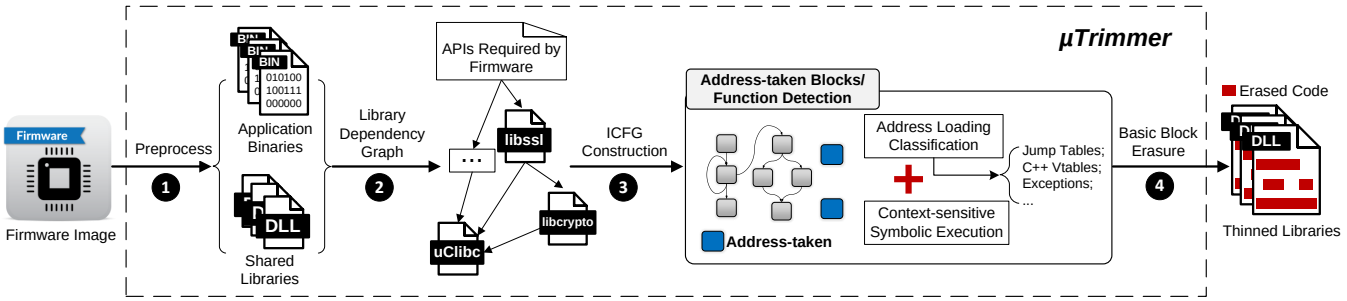

Figure 1: The overview of the μTrimmer. The whole process consists of four steps.

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.

3.1 Overview

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

1. Preprocess Given a firmware image, we adopt Binwalk [24] 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 [25].

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 [26–28] 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. The following subsections present our inter-procedural control flow graph (ICFG) construction.

3.2 ICFG Construction

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) [25] 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 [12, 23] 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.

3.3 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, 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). Function calls using pointers (e.g., callback functions) also belong to this type. Type ❷ and Type ❸ are corresponding to jump tables and C++ virtual functions,

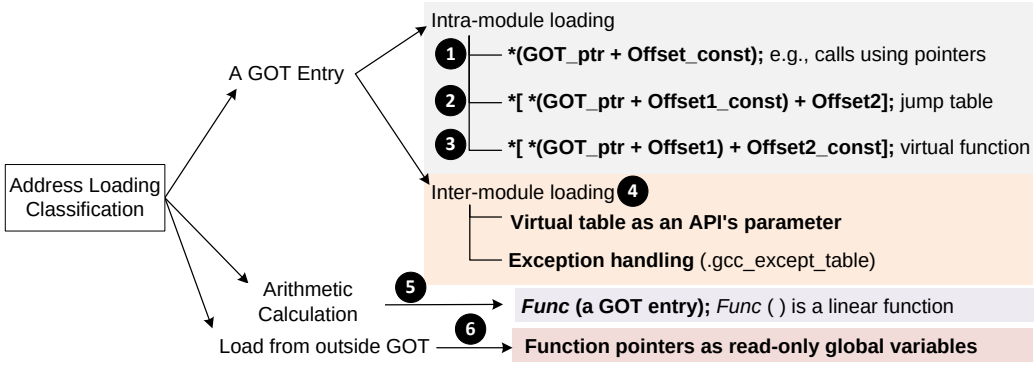

Figure 2: Six address loading patterns according to how indirect control flow targets are loaded from memory.

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

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 of inter-module address loading, in which a module's GOT can be accessed by a different module's instructions. Due to the lack of a global view, the AT function detection in one module 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 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 [12] only covers Type ❶ and Type ❷—missing any type could lead to incorrectly removing used code.

3.4 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. Interested readers are referred to our full paper in ASPLOS 2022 [29] for detailed detection methods.

4 RESULTS & CONTRIBUTIONS

We build μTrimmer on top of angr [30] 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 (TP-Link Archer A10). 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 [8, 12] as an upper bound of library debloating; μTrimmer's debloating capability competes with the static linking results and outperforms piece-wise [8] and Nibbler [12]. Interested readers are referred to our full paper in ASPLOS 2022 [29] for detailed evaluation data. Next, 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.

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 [31]. 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 3 shows

Table 3: 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%

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 2), while static linking can correctly remove functions in the ".data.rel.ro" section.

Directly comparing related work [8, 12, 13] is infeasible because of their specific assumptions (e.g., source code and runtime support) and different platform requirements. Fortunately, both piece-wise [8] and Nibbler [12] 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.

Contributions Overall we make 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. μTrimmer's debloating capability is on a par with the static linking's code downsizing results.

Research Impacts This study leads to one first-author publication in ASPLOS 2022 [29], and this paper also passed Artifact Evaluation and obtained three ACM badges: Artifact Available, Artifact Functional, and Results Reproduced. In addition, this study leads to a research funding from Cisco: "Security Hardening of IoT Devices via Debloating Shared Libraries." The potential customers of our proposed solution are individuals and companies who want to secure embedded systems via automated binary hardening.

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

REFERENCES

[1] J. Sattler, "Attack Landscape H2 2019: An Unprecedented Year for Cyber Attacks." http://tiny.cc/esj1tz, March 2020.
[2] Y. David, N. Partush, and E. Yahav, "FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware," in *Proc. ASPLOS*, 2018.
[3] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal War in Memory," in *Proc. S&P*, 2013.
[4] C. Details, "Security Vulnerabilities (Memory Corruption)." https://www.cvedetails.com/vulnerability-list/opmemc-1/memory-corruption.html, [online].
[5] G. Pacchiella, "CVE-2020-8423: Exploiting the TP-LINK TL-WR841N V10 Router." https://ktln2.org/2020/03/29/exploiting-mips-router/, March 2020.
[6] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-Oriented Programming: Systems, Languages, and Applications," *ACM TOPS*, vol. 15, March 2012.
[7] E. Andersen, "uClibc is a small C standard library intended for Linux kernel-based OS on embedded systems and mobile devices." https://www.uclibc.org/, [online].
[8] A. Quach, A. Prakash, and L. Yan, "Debloating Software through Piece-Wise Compilation and Loading," in *Proc. USENIX Security*, 2018.
[9] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares," in *Proc. NDSS*, 2014.
[10] D. D. Chen, M. Egele, M. Woo, and D. Brumley, "Towards Automated Dynamic Analysis for Linux-based Embedded Firmware," in *Proc. NDSS*, 2016.
[11] B. Feng, A. Mera, and L. Lu, "P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling," in *Proc. USENIX Security*, 2020.
[12] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Nibbler: Debloating Binary Shared Libraries," in *Proc. ACSAC*, 2019.
[13] C. Porter, G. Mururu, P. Barua, and S. Pande, "BlankIt Library Debloating: Getting What You Want Instead of Cutting What You Don't," in *Proc. PLDI*, 2020.
[14] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello, "Shuffler: Fast and Deployable Continuous Code Re-Randomization," in *Proc. OSDI*, 2016.
[15] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-Flow Integrity: Precision, Security, and Performance," *ACM CSUR*, vol. 50, April 2017.
[16] Y. Sui and J. Xue, "SVF: Interprocedural Static Value-Flow Analysis in LLVM," in *Proc. CC*, 2016.
[17] M. Jiang, Y. Zhou, X. Luo, R. Wang, Y. Liu, and K. Ren, "An Empirical Study on ARM Disassembly Tools," in *Proc. ISSTA*, 2020.
[18] The Santa Cruz Operation, "System V Application Binary Interface MIPS RISC Processor Supplement, 3rd Edition." https://refspecs.linuxfoundation.org/elf/mipsabi.pdf, February 1996.
[19] GCC Manual, "Options for Code Generation Conventions." https://gcc.gnu.org/onlinedocs/gcc/Code-Gen-Options.html, [online].
[20] G. Balakrishnan and T. Reps, "WYSINWYX: What You See is Not What You eXecute," *ACM TOPLAS*, vol. 32, Aug. 2010.
[21] N. Redini, R. Wang, A. Machiry, Y. Shoshitaishvili, G. Vigna, and C. Kruegel, "BinTrimmer: Towards Static Binary Debloating Through Abstract Interpretation," in *Proc. DIMVA*, 2019.
[22] J. Lin, L. Jiang, Y. Wang, and W. Dong, "A Value Set Analysis Refinement Approach Based on Conditional Merging and Lazy Constraint Solving," *IEEE Access*, vol. 7, 2019.
[23] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-Flow Integrity Principles, Implementations, and Applications," *ACM TISSEC*, vol. 13, November 2009.
[24] R. Labs, "Binwalk: Firmware Analysis Tool." https://github.com/ReFirmLabs/binwalk, [online].
[25] X. Meng and B. P. Miller, "Binary Code is Not Easy," in *ISSTA*, 2016.
[26] A. Altinay, J. Nash, T. Kroes, P. Rajasekaran, D. Zhou, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, C. Giuffrida, H. Bos, and M. Franz, "BinRec: Dynamic Binary Lifting and Recompilation," in *Proc. EuroSys*, 2020.
[27] D. Williams-King, H. Kobayashi, K. Williams-King, G. Patterson, F. Spano, Y. J. Wu, J. Yang, and V. P. Kemerlis, "Egalito: Layout-Agnostic Binary Recompilation," in *Proc. ASPLOS*, 2020.
[28] X. Meng and W. Liu, "Incremental CFG Patching for Binary Rewriting," in *Proc. ASPLOS*, 2021.
[29] H. Zhang, M. Ren, Y. Lei, and J. Ming, "One Size Does Not Fit All: Security Hardening of MIPS Embedded Systems via Static Binary Debloating for Shared Libraries," in *Proc. ASPLOS*, 2022.
[30] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *S&P*, 2016.
[31] Red Hat Customer Portal, "Static Linking Not Supported in Red Hat Enterprise Linux 8." https://access.redhat.com/articles/rhel8-abi-compatibility, May 2019.