Guided Feature Identification and Removal for Resource-Constrained Firmware

RYAN WILLIAMS, Northeastern University, USA
TONGWEI REN, Worcester Polytechnic Institute, USA
LORENZO DE CARLI, Worcester Polytechnic Institute, USA
LONG LU, Northeastern University, USA
GILLIAN SMITH, Worcester Polytechnic Institute, USA

IoT firmware oftentimes incorporates third-party components, such as network-oriented middleware and media encoders/decoders. These components consist of large and mature codebases, shipping with a variety of non-critical features. Feature bloat increases code size, complicates auditing/debugging, and reduces stability. This is problematic for IoT devices, which are severely resource-constrained, and must remain operational in the field for years.

Unfortunately, identification and complete removal of code related to unwanted features requires familiarity with codebases of interest, cumbersome manual effort, and may introduce bugs. We address these difficulties by introducing PRAT, a system which takes as input the codebase of software of interest, identifies and maps features to code, presents this information to a human analyst, and removes all code belonging to unwanted features. PRAT solves the challenge of identifying feature-related code through a novel form of differential dynamic analysis, and visualizes results as user-friendly *feature graphs*.

Evaluation on diverse codebases shows superior code removal compared to both manual feature deactivation and state-of-art debloating tools, and generality across programming languages. Furthermore, a user study comparing PRAT to manual code analysis shows that it can significantly simplify the feature identification workflow.

CCS Concepts: • **Security and privacy** → **Software security engineering**; • **Software and its engineering** → *Software usability*.

ACM Reference Format:
Ryan Williams, Tongwei Ren, Lorenzo De Carli, Long Lu, and Gillian Smith. 2021. Guided Feature Identification and Removal for Resource-Constrained Firmware. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2021), 25 pages. https://doi.org/10.1145/3487568

1 INTRODUCTION

Like most modern applications, firmware for IoT devices generally includes a significant amount of third-party open-source software components. Code reuse has many advantages, as it speeds up the development process and simplifies the deployment of complex functionality. However, it also comes with drawbacks. For a firmware developer, including third-party components implies bringing significant complexity into their codebase. Mature implementations of popular

Authors' addresses: Ryan Williams, Northeastern University, Boston, USA; Tongwei Ren, Worcester Polytechnic Institute, Worcester, USA; Lorenzo De Carli, Worcester Polytechnic Institute, Worcester, USA; Long Lu, Northeastern University, Boston, USA; Gillian Smith, Worcester Polytechnic Institute, Worcester, USA.

functionality—for example, publisher/subscriber communication and multimedia encoding/decoding—include a myriad of features, not all necessarily relevant to every deployment scenario. This is an instance of a phenomenon common in software development, where codebases over the years grow larger and larger due to feature creep [41]. A representative example is Mosquitto, an IoT-oriented middleware that implements the popular publisher/subscriber MQTT protocol. A review of its codebase reveals 19 discrete, optional features in addition to the core MQTT functionality. Several of them (e.g., Websockets support) are relevant to very specific use cases and can be safely dropped in other scenarios.

While abundance—or overabundance—of features is not a problem *per se*, it comes with negative consequences. Feature bloat generates larger binaries which in turn consume precious hardware resources, especially on resource-constrained IoT devices. Furthermore, uncommon features tend to receive little scrutiny and—when exposed—have long been used for attacks. A well-known example is the inclusion of outdated ciphers in TLS [18, 19]. Finally, smaller codebases are more easily audited and vetted for bugs. Vetting is particularly important in the IoT domain, where devices may operate in the field for years without updates [45]. For these reasons, it is critical to minimize the size and functionality of third-party libraries integrated in IoT firmware, by removing unneeded features and their corresponding code.

Unfortunately, manual feature identification and removal is a daunting task for codebases which consist of tens or hundreds of thousands of lines of code, particularly if a given project must include multiple of such codebases. Automating the task is also difficult, as none of the debloating and feature location techniques in literature can straightforwardly be applied to the task at hand. Some [22, 26, 28, 37, 49] rely on the availability of test cases to identify features. Generating such test cases is cumbersome for programmers (especially if unfamiliar with the codebases) and we found that, in practice, not every codebase ships with suitable test cases. Others [20, 24, 29, 30, 36, 40, 52] rely on domain expertise or externally-provided knowledge bases (e.g., through web mining or code annotations), which are unlikely to be readily available for every program of interest.

In this paper, we tackle these issues by proposing PRAT (**PR**ogram feature **A**nalysis **T**oolkit): a novel approach to feature discovery and removal in third-party code. At a high level, PRAT conducts automated feature discovery based on the analysis of build configurations. It then performs differential dynamic analysis to map each feature to the related lines of code and presents each mapping to the analyst in a concise, easy-to-understand graphical form. Once the analyst selects features for removal, all related lines of code are pruned, and the result is tested for correctness.

Designing an effective approach to feature removal entails tackling several challenges. First, features must be discovered without user assistance, which we achieve by deriving *hints* from build system flags. This is based on the fact that in many cases developers expose discrete, optional functionality as build-time options. Interestingly, however, our analysis (Section 8) finds that even if a feature is selectively disabled at compile-time via build flags, a significant amount of feature-related code (up to 29% of LOCs in our experiments) may still end up in the compiled binary! This is because manual deactivation of a feature simply excludes source files or libraries from compilation while leaving related functionality in shared files untouched. This functionality is interleaved with useful code and is in general not identifiable via dead code elimination. In order to ensure that we identify all feature-related code, we devise a new code pruning algorithm based on differential coverage analysis. Our algorithm executes the code of interest on a variety of test cases, to identify lines of code related to each build-exposed feature. To ensure that our approach works even for codebases that ship without high-quality test cases (or none at all), we introduce automated, high-coverage test case generation via symbolic execution. If the analyst deems a particular feature unnecessary, *all* related lines of code can thus be removed.

Another issue is that, while the problem of unused code removal has been studied in recent years (e.g., [38, 48]), allowing an analyst to understand and control the process has received much less attention. Conveying the set of available

features, and the ramifications of each feature in the codebase is complex but crucial to enable the analyst to make informed decisions. To solve this, we propose the novel formalism of *feature graphs*, to represent the complex dependencies between program features and their implementation. We also evaluate the usability of PRAT as a code understanding tool, by conducting a user study contrasting manual and PRAT-based analysis of code features.

PRAT's initial design focuses on C/C++ codebases using Make or cMake, due to the popularity of these languages and build systems in the IoT/embedded space. In order to validate this decision, we ranked GitHub projects by popularity (based on March 2021 data) and retained the 30 most popular project which are IoT-related in nature (i.e., explicitly target the embedded and/or IoT domains). We then determined language and build system for each of them; 18 out of 30 (60%) were Make/cMake-based and developed in C/C++. However, we also emphasize that PRAT's design is language-agnostic, requiring only the ability to compute code coverage and perform symbolic execution on the compiled program where there is a need for symbolically-generated test cases. Supporting a specific codebase requires the ability to parse that codebase's build system scripts, and the availability of tooling to compute code coverage (and additionally to perform symbolic execution if there is a need for symbolically-generated test cases). In practice, we found that implementing support for a language/build system is a low-effort task; to demonstrate this, we extended our prototype beyond C/C++ codebases by supporting Rust code using the Cargo build system.

Overall contributions:

(1) We present PRAT, a novel tool for the identification and removal of features from a program. Feature identification is based on the analysis of build system configurations, while feature removal uses a novel dynamic analysis algorithm leveraging comparison of code coverage data.
(2) We propose the use of *feature graphs* to represent available program features and their implementation within a protocol implementation's source code.
(3) We evaluate PRAT on diverse codebases, showing that it removes significantly more code than manual analysis and state-of-the-art automated tools, w/o introducing bugs.
(4) We demonstrate generality by applying PRAT to codebases developed in different languages (C, C++, and Rust) and using different build systems (Make, cMake, autoconf, and Cargo).
(5) Through extensive fuzzing and symbolic testing, we verify that PRAT does not introduce unexpected bugs.
(6) We conduct a user study comparing feature identification via manual analysis and PRAT. Results suggest that using PRAT can result in a streamlined workflow for a developer performing feature identification tasks.

Our PRAT prototype's source code is publicly accessible; see Section 14 for details.

2 CASE STUDIES

In this section, we motivate our work by reviewing the issue of software bloat w.r.t. various IoT-relevant codebases.

2.1 Communication Protocols

Smart, embedded appliances such as cameras, locks, thermostats, and a variety of sensors, are oftentimes equipped with the ability to connect to other devices and the Internet. For example, temperature sensors may be connected in a wireless network to collect temperature information in a large building automatically. Designers use a variety of low-overhead, application-level network protocols for communication between embedded devices; examples include MQTT, AMQP, and DDS. Many of these protocols have robust and well-maintained open-source implementations, for example, Mosquitto [1]

Name	Description	#Features
Mosquitto [1]	Implementation of MQTT: a publisher-subscriber messaging protocol based on brokers	19
azure-uamqp-c [2]	Implementation of AMQP: a protocol for messaging queuing and routing, supporting both publish-subscribe and point-to-point communications	16
OpenDDS [9]	Implementation of DDS: a publish-subscribe messaging protocol supporting discovery and quality-of-service	9
libzmq [10]	Implementation of ZeroMQ: a message-oriented middleware supporting both broker-less and broker-based communications	20
LoRaMac-node [7]	Reference implementation of LoRaWAN stack for low-power wide-area networks	12
LibCoAP [8]	Implementation of CoAP: a REST-based protocol for resource-constrained devices	8
LibNyoci [4]	Implementation of CoAP	10
Quiche [12]	Rust implementation of the QUIC transport protocol	4
FFmpeg [14]	Collection of audio/video codecs and algorithms	32
rav1e [15]	Fast and safe AV1 encoder written in Rust	14
libaom [11]	Alliance for Open Media video codec implementation	20

Table 1. Examples of relevant programs and their respective number of user-configurable features

for MQTT, and OpenDDS [9] for DDS. We also include Quiche, a userspace implementation of the QUIC transport protocol and HTTP/3, due to the current industry interest in applying QUIC to the IoT space [25].

Features. Table 1, lines 2-9, summarizes a set of popular protocol implementations in the IoT space. For each, we manually inspected source code and determined the number of optional features (column 3). Table 1 corroborates our insight that IoT-related software comes equipped with a significant amount of non-core functionality. For example, Mosquitto implements support for Websockets, which simplifies communication between a web browser and a server. If an IoT installation does not include a browser UI, the feature—whose implementation consists of 800+ LOCs—can be deactivated.

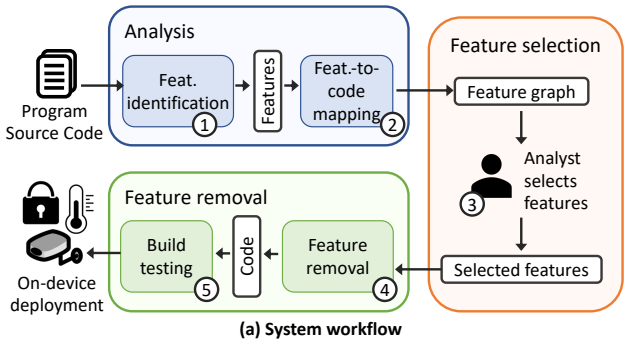
2.2 Multimedia Encoders/Decoders

Many IoT sensors incorporate media capturing and streaming capabilities. Audio/video codecs are extremely complex pieces of software and reuse of standardized implementations is common; we therefore consider leading open-source implementations of multimedia codecs and streaming logic as a relevant case study. We include three popular and actively maintained codebases: FFmpeg [14], an extensive, multiplatform collection of audio/video codecs and related tools; and two implementation of the industry-standard AV1 video codec: rav1e [15] (written in Rust), and libaom [11] (C/C++).

Features. Based on lines 10-12 of Table 1, the three programs in this category include, in aggregate, 66 discrete build-exposed features. This is not surprising as media tools tend to implement support for optional optimizations, algorithms, and subcomponents (e.g., video post-processing routines in FFmpeg). This result underlines the importance of customizing these codebases for embedded applications.

2.3 Motivation of Our Work

The abundance of optional features, and the difficulty of fully removing them, are undesirable. First, when building firmware for highly resource-constrained IoT devices, it is essential to avoid unnecessary use of storage space for useless functionalities. Feature-level removal goes beyond dead or unreachable code and is focused on reachable code that is



Mosquitto features	
TLS	SSL/TLS support
systemd	Systemd notifications
Bridge	Inter-broker connections
Wrap	tcpd/libwrap support
Persistence	Message persistence
Websockets	Websockets support
Memory_tracking	Mem. usage tracking
TLS_PSK	TLS/PSK support
EPOLL	Use `epoll()` syscall
. . .	

(b) Mosquitto features (some omitted for clarity)

Fig. 1. System overview

not used or needed. It is worth pointing out that leftover code may also offer attackers additional avenues (larger attack surface) to mount attacks, such as ROP and code reuse [42].

Second, optional features present opportunities for vulnerabilities to lurk in a codebase. To validate this, we performed a survey of CVE reports [13] related to a subset of programs in Table 1 (Mosquitto, azure-uamqp-c, OpenDDS and FFmpeg). We sampled approximately 100 results, and eliminated those which did not apply to feature-related functionality based on description. For the remaining ones, we identified the lines of code where the vulnerability lies, and confirmed that these lines are removed by PRAT when disabling the relevant feature. This analysis resulted in eight CVEs: CVE-2018-12546 in Mosquitto, CVE-2019-1535 in OpenDDS, and CVEs 2018-13303, 2017-16840, 2019-9721, 2019-9718, 2019-11339, 2018-1999015 in FFmpeg. Five of those are rated “medium”, two “high”, and one “critical”, for an average CVSS score of 7.3. While these features can be manually deactivated, PRAT simplifies the process of discovering and disabling them, encouraging and streamlining their removal.

Code removal (or debloating) at the feature-level directly answers the need for methods to keep IoT firmware small in code size and minimize attack surface. It also facilitates static and dynamic analyses. Refining the code by removing all unnecessary features makes analysis scope smaller, complexity lower, and precision higher. Compared with feature-unaware debloating, feature-level debloating is more user-friendly, explainable, and tractable. It is easier for users to understand what is being removed and why; it guides and takes inputs from the analyst through the debloating process.

We believe that an easy-to-use IoT feature-based software debloating tool is an important—yet missing—building block towards optimized, reliable IoT firmware. Our core goals are therefore to: (i) identify available features, and their ramifications within a codebase, (ii) ensure that deactivation of a feature at build time results in removal of all code solely associated with that feature, and (iii) ensure the process results in functionally correct binaries.

3 SYSTEM WORKFLOW

In this section, we demonstrate PRAT via a concrete use case: feature pruning of the popular MQTT broker, Mosquitto. Sections 4-7 describe each step in detail.

3.1 Mosquitto Overview

Mosquitto is a popular implementation of the MQTT publisher/subscriber communication protocol. It is a mature codebase that has incorporated several features over time, summarized in Figure 1(b). In general, not all such features are necessary for a given deployment. In our scenario, we assume an analyst uses our tool to prepare a Mosquitto build for a Linux-based sensor network. Devices have limited storage which must suffice for the OS and additional custom code. The network

exists in a controlled power generation building, and it is not connected to the Internet. The function is non-critical (environmental monitoring rather than process control) but access to the network nodes is difficult (most of the monitoring network exists within a reactor containment building) so software reliability is important. The process of feature selection is carried in 5 steps, described in the following and depicted in Figure 1(a).

3.2 Step 1: Feature Identification

First, PRAT identifies all features available within the source code. Feature information is not readily available or always documented, and therefore PRAT infers features by analyzing and filtering the build system options. We implement parsers to analyze the build system of the project and return a list of available features.

We term the remaining set of build options as the *candidate feature set*—in other words, those are options that are likely to control activation/deactivation of optional protocol features. As part of the feature identification step, we also have a process that filters out certain features from being displayed to the analyst. The default behavior—and that used in our running example—is to display all features to the analyst so they can determine which are important or not. However, if we want to simplify the process for the analyst further, we filter out features that would otherwise inundate the analyst with options. The feature identification step is described in greater detail in Section 4. In our example, the tool would identify the set of features in 1(b).

3.3 Step 2: Feature-to-Code Mapping

In the second step, PRAT identifies lines of code exclusively associated with each feature (i.e., lines of code whose purpose is to implement a given feature but are not associated with/necessary for any other functionality). This is necessary since, as outlined in Section 1, simply deactivating a build option associated with a feature does not guarantee full removal of all associated code. The process builds a set of binaries, each with a different build option—among those identified in step 1—turned off. All binaries are instrumented to collect *code coverage data* and executed on a set T of symbolically-generated tests. Differential analysis of code coverage allows PRAT to determine which code is associated exclusively with each feature. Because all the binaries are instrumented to track coverage information when we run a test against the binary, all code that gets executed is marked in corresponding coverage files. Seeing code being executed with one feature activated but not another means that the code block pertains to the feature being tested. Results of these tests are gathered and used to determine the lines of code associated with each feature (the process is described in detail in Section 5. The feature-to-code mapping is then converted to a graphical representation we call a *feature graph*. For a given source tree, a feature graph lists high-level features and associates each with source files and individual LOCs. Figure 2 shows a simplified view of the feature graph for Mosquitto's *Websockets* feature.

3.4 Step 3: Feature Selection

Feature graphs—described in detail in Section 6—constitute an easy-to-use decision-support tool for feature selection. Using a feature graph, the analyst: (i) identifies the set of available optional features; (ii) assesses the amount of code associated with each; and (iii) delves into the specifics of each feature's implementation. As the output of this step, the system receives back a set R of features that the analyst wishes to be fully removed from the code. In our example, the analyst identifies *Websockets* and possibly *TLS*, *TLS_PSK* as superfluous (browser connectivity is not required, and the analyst decides that, given the controlled setting, encryption may be unnecessary and even hamper debugging).

3.5 Steps 4-5: Feature Removal and Testing

PRAT removes from the source tree all lines of code associated with the features in R. The modified source is then compiled into the final binary. The three identified features in our example still have code artifacts remaining in the source code when deactivated via build options. By using our tool, on average we remove an extra 46,396 LOCs, which amounts to an additional 2.9 MB reduction in the size of the compiled binary. The program binary is once again run through the set T of test cases as well as rounds of fuzzing to ensure its functional correctness. We note that the extensive use of dynamic analysis, both in feature identification and testing, comes with some limitations. We further discuss these limitations, and motivate why we deem them acceptable, in Section 10.

4 FEATURE IDENTIFICATION

First, PRAT identifies a set of discrete, high-level features that can be selectively deactivated without breaking a software's core functionality. While many possible definitions of features are used in literature, we converged on one which is both practical and relevant to the user. Often, build options are used by developers to enable activation/deactivation of various pieces of functionality, which are important but not needed by all users of a software. This is a common practice in software development, and we use for our definition:

DEFINITION. *A feature is a set of lines of code which can be selectively activated or deactivated by operating on a single build configuration option.*

This definition has the advantage of encompassing portions of the source code which have been implicitly marked as optional—by defining them as toggleable via build-time options—by the software developer. This approach fits well with our workflow, as our goal is to operate on discrete features which do not have the potential to break core functionality of the program. Indeed, by definition none of the features which are exposed via the build system are necessary, and they can freely be removed should the analyst decide to do so.

It should be noted, however, that there may be additional code related to such a feature that remains in the source even when the related build option is turned off: the code associated with a build option generally represents a subset of the code that can be removed if the related feature is not needed. Indeed, our evaluation in Section 8 shows that—somewhat counter-intuitively—a significant percentage of feature-related code may remain even when the relevant feature is disabled at build time.

Given these considerations, identifying features begins by analyzing build configuration files. Such analysis must, by necessity, be based on heuristics, which encode expert understanding of how configurations are created and used. We built specialized analyzers for two of the most popular build systems for C/C++, Make/cMake, and autoconf, and Cargo for Rust:

cMake. The analyzer builds the root project in the source tree using `cmake -LA | grep BOOL`, which returns a list of toggleable options. We posit that a feature which can only assume boolean values *true* or *false* corresponds to a user-defined feature (further post-processing is applied; see below).

Autoconf. The analyzer obtains a list of build-time options by executing `configure -help`, and retains those whose description includes the words "feature" or "optional".

Cargo. The analyzer parses `Cargo.toml` to find any defined features and returns a list of those that are non-default.

Discussion. A possible issue is that of false positives. Build configurations typically intermix options related to high-level functionality with ones that control low-level aspects of the compilation, which are not directly related to lines of code (e.g., shared library location, optimization level). Most of these spurious options are standard across codebases and are automatically discarded by our parser. We further apply filters to hide features that are meant for debugging or other developer-specific options. These features are identified using simple heuristics about naming conventions for in-line directives.

Another issue is false negatives, since relevant features may exist that are not exposed to the build system. However, a detailed analysis of representative codebases (discussed in Section 8.2) did not yield any such feature, suggesting that PRAT's feature discovery is robust.

The output of the feature discovery step is a set $F = \{f_1, ..., f_n\}$ of n discovered features.

Algorithm 1 Feature-to-code mapping

Input: Program source P
Input: Set $F = \{f_1, ..., f_n\}$ of features in P
Input: Set U of provided unit tests (if any)
Output: Set $\mathcal{D} = \{D_1, ..., D_n\}$, where D_i is the set of LOCs associated to feature $f_i \in F$

$\mathcal{D} \leftarrow \emptyset$;
$S \leftarrow$ SymbolicTestGeneration(P);
$T \leftarrow U \cup S$;
$B_{all} \leftarrow$ Compile(P);
$L_{all} \leftarrow$ CoverageAnalysis(B_{all}, T);
for each $f \in F$ **do**
    $P_f \leftarrow$ DisableFeature(P, f);
    $B_f \leftarrow$ Compile(P_f);
    $L_f \leftarrow$ CoverageAnalysis(B_f, T);
    $D_f \leftarrow L_{all} \setminus L_f$;
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{D_f\}$;

5 FEATURE-TO-CODE MAPPING

To ensure full removal of feature-related code from the target program, once the set F of features is available, PRAT proceeds to identify the subsets of source code that are exclusively associated with each feature. We perform this mapping at the granularity of individual lines of source code (LOCs in the following). We find that working with LOCs has two important advantages: (i) it is sufficiently fine-grained to be useful in practice, and (ii) it makes analysis results easily readable and understandable to a human analyst.

Our mapping algorithm uses dynamic analysis. The core idea is to compare *code coverage* achieved by instrumenting, compiling, and executing the same binary with and without a particular feature. After mapping back the executed binary instructions to the respective LOCs, LOCs that are only executed when a given feature is active are inferred to be exclusively related to that feature.

Like any dynamic analysis approach, the correctness of our algorithm is crucially dependent on exploring as many different code paths as possible. Therefore, we give considerable attention to accruing a comprehensive set of test cases, using symbolic execution to extend the unit tests that are typically distributed with a program's source code. The rest of this section describes our code coverage analysis and symbolic test generation. The overall procedure is outlined in Algorithm 1.

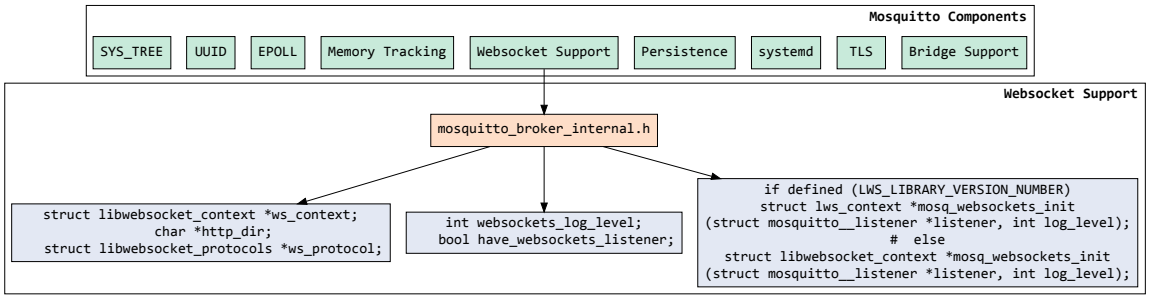


Fig. 2. Simplified feature graph example (from Mosquitto)

5.1 Per-feature Builds

Given the set of features F, PRAT builds $n + 1$ separate versions of the target program binary. First, our system builds a version of the binary with *all* features enabled (line 4 of Algorithm 1). Next, it builds additional n versions where the $i - th$ build has all features enabled except feature f_i (lines 6-8 in Algorithm 1). This is typically the most time-consuming step in the feature removal process, as it requires re-compiling a potentially large codebase multiple times. Overall, the entire PRAT analysis pipeline takes on average 12.8 minutes to complete on the codebases evaluated in Section 8. We consider this acceptable as manually identifying all features in an unknown codebase, and tracking all the locations in the code where each feature is implemented, is likely to take significantly longer. Additionally, tools like ccache [3] could be applied to further speed up the process via memoization. At this stage, we also discard build options that result in a failed compilation. This step outputs a set of $n + 1$ binaries $B_{all}, B_1, ..., B_n$, where B_{all} has all features enabled, and each binary B_i has all features enabled except f_i.

5.2 Code Coverage Analysis

The next step (lines 5 and 9-11 in Algorithm 1) leverages *code coverage analysis* to identify lines of code associated with each feature via an exclusion process. Code coverage instruments compiled binaries to determine which lines in the source code correspond to binary instructions that were executed in a given run. Consider the set L_i of lines of code returned by the tool when running B_i, i.e., the binary with feature f_i disabled. We posit that lines in $D_i = L_{all} \setminus L_i$ (where L_{all} is the set of lines of code returned with all features enabled) represent the logical implementation of feature f_i.

5.3 Symbolic Test Case Generation

Code coverage, to be successful, requires the availability of tests that make use of as many features as possible. We rely on a set of test cases T, to provide the highest-fidelity results.

First, this set includes the set of unit tests U shipped with the code, if any. For example, both azure-uamqp-c and OpenDDS ship with a comprehensive suite of unit tests which we used without modifications. However, other software tools (e.g., Mosquitto) may not ship with sufficiently comprehensive tests.

In order to ensure that the analysis can proceed regardless of the availability of U, we leverage symbolic execution to generate an additional set S of high-code-coverage tests.

It should be noted that, in general, symbolic execution is not the only approach that can be used to carry such task, however we found it to be the most effective one in our domain. Beyond symbolic execution, we investigated using state-of-the-art testing tools, namely the concolic executors QSYM [50] and Driller [46]. However, both are based on

Parameter	Option(s)
`libc`	uclibc
`runtime`	posix-runtime
`sym-args`	0 3 4
`sym-files`	2 4
`max-fail`	1
`max-time`	60

Table 2. General parameters used for test case generation via KLEE

AFL [17], which is not designed for fuzzing network protocols (network protocol implementations constitute some of the most relevant targets for our analysis in the IoT space). Integrating AFL file-based interface with network code requires either modifying the target program in non-trivial ways or engineering complex test harnesses [27]; while network-specific forks of AFL have been developed [16], they are not integrated with concolic engines. Furthermore, we want the executor to generate a single test case for each new path explored in the codebase, which we can then simply replay against any variants of the same implementation.

Based on the considerations above, we leverage the KLEE symbolic execution engine [21] to generate high-quality tests. KLEE is a symbolic virtual machine built on top of the LLVM compiler infrastructure, which aptly suits our needs. In particular, we use its capability to generate a test case for every new path that it explores. By running KLEE against a target, we can generate a suite of high-coverage test cases, S, even in the absence of bundled unit tests. Overall, the final set T of tests consist of $T = U \cup S$.

The test generation process is summarized in lines 2-3 of Algorithm 1. In particular, the SYMBOLICTESTGENERATION procedure in line 2 consists of the following steps:

(1) Compile source code to LLVM bytecode
(2) Run KLEE on the bytecode with a given set of parameters. When KLEE explores a new path, it generates a test case with the concrete inputs needed to traverse that path.

During code coverage analysis, KLEE-generated tests can be replayed against the actual protocol binary by using the `klee-replay` utility. This approach requires no modification of source code; the analyst can set the number (and corresponding size) of the symbolic variables and files that KLEE uses. To balance our tradeoff between runtime for test generation and code coverage achieved through our tests, we set parameters for `max-fail` and `max-time`. We run KLEE against our target protocols for 60 minutes and, on average, generate 4,369 tests.

5.4 KLEE Configuration and Alternatives

Running KLEE requires setting arguments for the symbolic environment. However, PRAT strives to automate the entirety of the testing process so it is opaque to the analyst. Through testing various permutations of candidate parameters to KLEE, we settled on a default parametrization (given in Table 2) that provided generally comprehensive results across different projects. If the operator desires to maximize code removal and has sufficient domain knowledge, it is possible to refine the set of symbolic arguments and files passed to the LLVM executable.

Given our domain knowledge of the protocols under test, we can run the LLVM executables with a targeted set of symbolic arguments and files. For example, if we want to exercise the functionality of starting a broker using a configuration file instead of command-line arguments, we can pass `-sym-files <NUM> <N>` where `NUM` is the number of files of size `N`. Using knowledge of the system under test, we can target KLEE much more precisely; however,

since this knowledge cannot be assumed, we run KLEE using a standard set of parameters that tend to result in generally comprehensive coverage. In all cases, our targets were tested using the same set of parameters, shown in Table 2.

We also explored the option of augmenting the base unit tests in U, by making some of their inputs symbolic and "amplifying" their coverage. This requires much more manual effort, as the analyst needs to understand how the code base and unit tests work in order to properly flag variables in the source as symbolic. Results in Section 8.3 suggest that this additional effort does not result in improved code coverage. Therefore, we do not consider this approach further.

Instead, to avoid requiring the user to manually modify the source code for symbolic execution purposes, we dismissed the idea of making individual variables symbolic for testing and instead left the target codebase untouched. This means that we run KLEE against our target binary with the aforementioned set of parameters without needing to manually insert any `klee_make_symbolic` calls. Utilizing KLEE in this way circumvents the need for manual instrumentation of the target codebase, and simplifies the overall workflow.

5.5 Correctness

By design, Algorithm 1 removes code conservatively, removing only those LOCs that are executed when the relevant feature is active, but not when the same feature is disabled. The algorithm **does not remove** LOCs that are never executed regardless of whether the feature is active or not. Thus, in the event that some feature-specific LOCs are not covered by the tests, they will not be removed through this process. This design purposely strives for soundness over completeness. It contains the effects of incomplete coverage, which thus results in the algorithm failing to remove all unused LOCs, but not in the removal of non-feature-related LOCs.

Overall, feature-to-code mapping outputs the sets $D_1, ..., D_n$ of lines of code associated with each feature $i = 1, ..., n$; and a *feature graph* (described in the next section).

6 FEATURE SELECTION

In the feature selection step, PRAT presents the results of feature identification and feature-to-code mapping to the analyst, who selects the features to be removed prior to deployment. A significant problem here is how to present the analyst with easy-to-understand and actionable information.

To address the problem, we devised the novel formalism of *feature graphs*. Despite its simplicity, we found this concept to be effective to quickly grasp both the set of available features and their footprint within a larger codebase. A feature graph is a directed acyclic graph (DAG) $F = \{V, E\}$. The set of vertices V includes nodes representing features, source files, and sets of lines of code. The set of edges E is constructed as follows. Consider a feature node $f \in V$, a source file node $s \in V$ s.t. the file s contains the implementation of f, and a set of n lines of code (also represented by nodes $l_1, ...l_n \in V$) which reside in s and are executed only when f is enabled. Then, we establish an edge (f, s). Furthermore, for each line of code of interest l_i, we establish an edge (s, l_i). In practice, contiguous lines of code are merged in a single node to limit the complexity of the graph. Informally, this results in a DAG where the roots are features, intermediate nodes are source files, and leaves are lines within source files. Figure 2 presents a simplified example of DAG, restricted to a single Mosquitto feature. In this case, the graph links Websocket support to the `mosquitto_broker_internal.h` header file and some of its content. Using the graph, a user can quickly descend from a feature into the source files where it is implemented, and—if necessary—examine the code implementing it. After completing the assessment, the analyst passes back to the system the set R of features marked for removal.

Program	#LOCs	#Source files	Size [MB]
Mosquitto [1]	79,859	953	27
azure-uamqp-c [2]	90,570	506	38
OpenDDS [9]	2,589,799	5,457	380
Quiche [12]	1,177,473	2,037	325
FFmpeg [14]	1,241,433	7,196	341
rav1e [15]	135,727	264	23
libaom [11]	452,822	1,158	273

Table 3. Characterization of codebases

7 FEATURE REMOVAL

In this step, PRAT receives the set of features to be removed, R, and the set of lines of code associated with each feature. It then performs feature removal by removing from the source the lines in $\bigcup_{i \in R} D_i$, and rebuilds the program binary. To ensure that the program continues to work correctly after feature removal, the system then re-runs the test suite, T, generated during feature-to-code-mapping and monitors the program's output for crashes and unexpected behavior.

8 EVALUATION

In this section, we evaluate PRAT in light of the following experimental questions:

- **Question #1: can PRAT identify all features within a codebase?** In section 8.2, we show that our feature analysis produces no false negatives and low false positives with no impact on the final binary.
- **Question #2: does our approach to test generation create tests that achieve high code coverage?** In Section 8.3 we show that test generation based on symbolic execution results in high code coverage, providing confidence in PRAT's dynamic analysis approach.
- **Question #3: does our feature removal approach lead to a measurable reduction in unused code?** In Section 8.4, we show that PRAT results in the removal of up to an additional **29%** lines of code, compared to the amount of code removed by manual deactivation of features at build time.
- **Question #4: how does our approach compare to related work in the domain of software debloating?** In section 8.6, we compare PRAT to Piece-Wise [38], a state-of-the-art debloating tool. PRAT leads to an additional significant reduction in code and binary size.
- **Question #5: do programs retain correctness after feature removal?** In section 8.7 we show the results of extensive testing via both symbolic tests and fuzzing on Mosquitto. Our analysis reached on average 84% code coverage without revealing any removal-related bugs.
- **Question #6: is our approach generalizable?** Sections 8.2 and 8.4 include evaluation of Rust codebases, demonstrating generalizability beyond C/C++. We further discuss support for multiple languages in PRAT in section 10.

We further conducted a user study to evaluate the effect of using PRAT on a developer's feature identification workflow. We separately discuss the study and its results in Section 9.

8.1 Implementation and Dataset

Tooling. To evaluate our approach, we created a prototype implementing the PRAT pipeline described in Sections 3-7. Internally, the prototype leverages `gcov` (for C/C++) and `kcov` (for Rust) to generate code coverage reports. The

Program	All features		No features (manual)		No features (PRAT)	
	Code size [LOC]	Binary size [B]	Code size [LOC]	Binary size [B]	Code size [LOC]	Binary size [B]
Mosquitto	18,983	1655112	18,345 (-3.4%)	1245648 (-24.7%)	14,273 (-25%)	498156 (-70%)
azure-uamqp-c	49,629	1250716	49,311 (-0.64%)	749564 (-40.1%)	35,293 (-29%)	701524 (-44%)
OpenDDS	207,936	47176232	206,514 (-0.68%)	46995256 (-0.38%)	202,498 (-2.6%)	39786816 (-15.7%)
Quiche	488,247	20237216	479,545 (-1.8%)	12714298 (-37.2%)	476,771 (-2.4%)	10533152 (-48%)
FFmpeg	1,122,317	30400928	945,890 (-15.7%)	26057568 (-14.3%)	674,620 (-40%)	19361728 (-36.3%)
rav1e	130,010	24480350	128,950 (-0.82%)	20382662 (-16.7%)	128,054 (-1.5%)	18060592 (-26.2%)
libaom	344,554	3517928	336,944 (-2.2%)	3444112 (-2.1%)	309,216 (-10.3%)	3188680 (-9.4%)

Table 4. Code reduction for target programs (% represent reduction in LOCs and bytes compared to “All features”)

prototype has been publicly released (ref. Section 14). To validate the correctness of feature removal on the Mosquitto broker codebase (Section 8.7), we built a fuzzing MQTT client based on the Boofuzz fuzzer [6]. We provide details on the fuzzer in Section 8.7.

Program Dataset. Table 3 describes the program codebases used for the experiments (large codebase sizes are due to the inclusion of tests and other resources). We selected these seven codebases after an extensive review of open-source middleware protocols and multimedia tools. Ultimately, we converged on codebases that represent stable, popular implementations of mature functionality relevant to the IoT space.

8.2 Feature Analysis

Methodology. In this section, we evaluate the effectiveness of PRAT in identifying optional features embedded in a codebase. We identify false positives and negatives by manually analyzing each codebase and collecting available features, and comparing the result with the list returned by PRAT. Manual identification of features was carried by two of the authors using the process detailed below. On average, each person spent 30 minutes per codebase. The union of these two sets of features was used as a reference for evaluating false positives and negatives. In order to collect features, we first obtained the list of build system flags, retaining those matching the definition of a feature described in Section 4. Further, to ensure that no hidden features (i.e., features not accessible to the build system) exist, we parsed and collected `#define` directives within the source code of all programs, manually comparing such variables with the identified features. This analysis did not reveal any features existing in the code but not accessible to the build system. Overall, our target codebases include 115 optional features. We were unable to support a total of 7 features across the targets; 2 of them were Windows-specific and thus incompatible with our framework, while 5 of them triggered various compilation bugs, which we determined to be unrelated to our infrastructure. The most interesting case was a bug we uncovered in azure-uamqp-c, where if no SSL implementation is specified to use, the code defaults to using OpenSSL. However, in the checks for the defined SSL implementation, in the unspecified, default case, the appropriate headers for OpenSSL are not imported. This results in compile-time errors, which were found through our feature extraction process. Following this discovery, we reported the bug to developers.

Results. In principle, feature identification could generate false negatives in two cases: (i) features configurable via the build system are not recognized; and (ii) features exists which are not exported to the build system. Mosquitto’s build system defines 20 build-time options, azure-uamqp-c 32, OpenDDS 10, Quiche 5, FFmpeg 34, rav1e 16, and libaom 26. Of those, respectively **19**, **16**, **9**, **4**, **33**, **14**, **20** are features according to our definition. Our feature analysis (ref. Section 4) identifies **all 115 true features**.

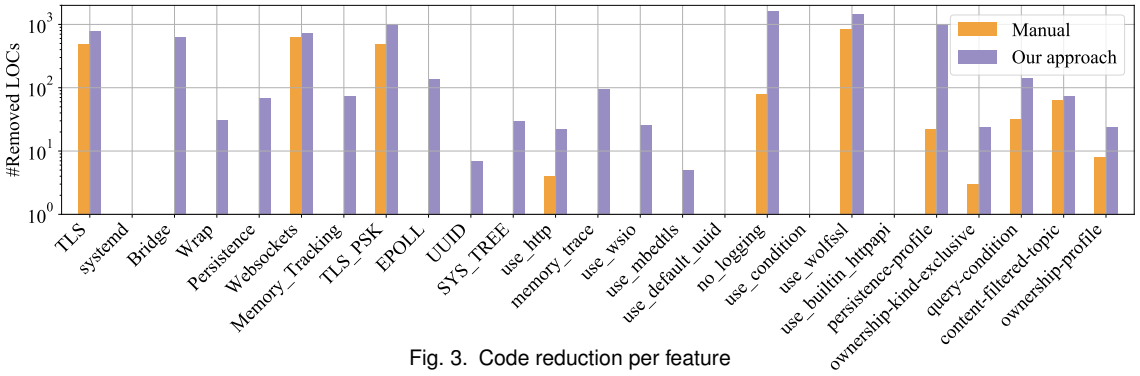


Fig. 3. Code reduction per feature

False positives can be generated if feature analysis confuses non-feature build options for features. In our analysis, this only happens in **1** case. The lone misidentified feature is the option to build with or without a shared library. However, this build option does not affect the program binary and has no source code associated with it; therefore, including it in the analysis does not affect the final result.

8.3 Symbolic Test Generation

Before evaluating feature removal, we briefly discuss PRAT's approach to generate test cases using symbolic execution. As in any dynamic analysis-based system, we rely on exercising as much code as possible. Comprehensive tests allow our approach to identify feature-related code accurately.

In our case, symbolic execution can be implemented in one of the two variants discussed in Section 5.3. First, KLEE can be directly run against our program binaries. Second, existing unit tests (if any) can be manually modified to turn some of the concrete inputs into symbolic ones. We evaluated both approaches. Running KLEE directly against the binary results, on average, in **80.2% LOC coverage**, while symbolizing existing tests results in **39.6% LOC coverage**. Direct binary execution via KLEE achieves superior coverage; therefore, we do not further consider the symbolization of existing unit tests.

8.4 Feature Removal: PRAT vs Manual Removal

Methodology. In this experiment, we compare the effectiveness of PRAT in removing unwanted feature code, compared to manual removal. For each codebase in our set, we do the following. First, for reference we compile each codebase to a binary with all features activated. Then, for manual feature deactivation, we conservatively assume that the user can manually identify and turn off all optional features (except those resulting in compilation bugs, as discussed in 8.2). After doing so, we compute both the #LOCs that actually get compiled, and the size of the compiled binary. Finally, we run PRAT on the codebase and configure it to remove *all* identified optional features, measuring both the resulting #LOCs and compiled binary size.

Results. Table 4 describes, for each program in our evaluation set, source code size, and compiled binary size for a version of the program with all features enabled (columns 2-3). It then compares code reduction obtained by manual feature deactivation (columns 4-5) and our approach (columns 6-7). Figure 3 provides a more in-depth look into the impact of our approach on the removal of individual features (for clarity, we randomly select 25 features across codebases). Note the log-scale y-axis; missing bars represent #LOC values of 0. In most cases, our approach results in a significant reduction

```
static void unpwd__free_item(struct
  mosquitto__unpwd **unpwd, struct
  mosquitto__unpwd *item) {
  mosquitto__free(item->username);
  mosquitto__free(item->password);
  mosquitto__free(item->salt);
  mosquitto__free(item);
}
```

Fig. 4. Example block of code which is not removed by toggling the TLS build-time option but is removed by PRAT (from `mosquitto/src/security_default.c`). The code implements a utility function which frees items in memory which correspond to a username/password. This is only enabled when using TLS, so PRAT removes the function when the TLS feature is disabled.

in the size of the program source compared to manual feature deactivation. Overall, PRAT resulted in up to **29%** more LOCs being removed compared to manual deactivation of feature-related options.

8.5 Characteristics of Code removed by PRAT

The code that is additionally removed by PRAT tends to look like what might typically be contained within a preprocessor conditional group. In other words, it is code whose execution is predicated upon the availability of a feature. Determining why such code was not marked by conditional preprocessor directives (e.g., `#ifdefs`) is beyond the scope of our work. However, we suspect that this may be related to the presence of numerous developers on the same project, which creates ambiguity wrt. the purpose of specific blocks code. In other words, developers may not single out those blocks as belonging to a feature, out of concern that core functionality may depend on it. Incidentally, this observation highlights the potential applications of PRAT as a code understanding tool, as PRAT can quickly infer and display the ramifications of a given feature with the codebase. An example of code removed by PRAT is shown in Figure 4.

8.6 Feature Removal: PRAT vs Piece-Wise

Methodology. In this section, we compare PRAT to a state-of-art debloating algorithm, Piece-Wise by Quach et al. [38]. Differently from PRAT, Piece-Wise is a binary debloating technique and focuses on unguided unused code removal. It works by generating a fine-grained program dependency graph and only loading code that is truly needed at runtime. In order to evaluate Piece-Wise, we manually de-activate all optional feature, compile the code, and run Piece-Wise on the resulting binary for additional code reduction. We execute PRAT using the same procedure as in Section 8.4.

It should be noted that the above is an imperfect comparison. Much like in the manual removal case, when using Piece-Wise the burden of manually discovering and disabling optional features is placed on the user. Conversely, PRAT performs this function with high accuracy and automatically, as elucidated in Section 8.2. It is also important to point out that PRAT can complement a binary debloating tool such as Piece-Wise, rather than replacing it. Feature removal at the source code and binary level work at different levels of abstraction. Therefore, each approach is able to identify "bloat" which is not visible to the other one. In our last experiment, we examine the advantages of the combined approach by performing the removal of all features via PRAT and further debloating the resulting binary with Piece-Wise.

Results. Table 5 compares the size of binaries generated by PRAT (column 3) with those debloated using Piece-Wise (column 4). We omit rav1e and Quiche, as they are written in Rust and Piece-Wise does not currently support this

Program	Binary size [B]			
	All features	PRAT	Piece-wise	Combined
Mosquitto	1655112	498156 (-70%)	569632 (-65.6%)	468080 (-71.7%)
azure-uamqp-c	1250716	701524 (-44%)	743492 (-40.6%)	701332 (-44%)
OpenDDS	47176232	39786816 (-15.7%)	40232032 (-14.7%)	38487792 (-18.4%)
FFmpeg	30400928	19361728 (-36.3%)	24632584 (-19%)	18796480 (-38.2%)
libaom	3517928	3188680 (-9.4%)	3189304 (-9.3%)	3187944 (-9.4%)

Table 5. Comparison of PRAT and Piece-wise (% represent reduction in bytes compared to "All features")

language. Binaries generated by PRAT are up to **17%** smaller than those generated by Piece-Wise, which validates the principle of feature-aware code removal as an effective debloating technique. Column 5 in Table 5 summarizes final binary size for each program in our dataset, when using both PRAT *and* Piece-Wise. Results show that using both tools in combination results in a modest improvement over PRAT alone, suggesting that it may be useful to combine source code- and binary-based debloating.

8.7 Correctness

Methodology. In this section we evaluate the correctness of PRAT's output using Mosquitto as a case study.

First, when mapping LOCs to a certain feature, PRAT must avoid false positives and minimize false negatives. To verify this, we parsed the output of code coverage analysis to generate code comparison reports which display, side-by-side, the original code and the code post-debloating, highlighting feature-relevant code. We then manually inspected these reports to identify inconsistencies.

Second, even when LOCs are correctly mapped to features and removed, it is important to ensure that the final compiled binary still works correctly. We evaluated the correctness of the codebase using a testing-based approach. First, we run available tests against the debloated codebase, which did not evidence any newly introduced bug. Second, we carry further analysis on Mosquitto by complementing existing tests with fuzzing. In particular, we built a custom MQTT fuzzing engines based on the popular Boofuzz fuzzer [6]. Fuzzing as a testing strategy was not considered or employed during the design and implementation of PRAT; therefore, we consider it a useful "sanity check" with potential to uncover issues not otherwise identified. To keep execution times practical, we did not test all possible combinations of features; rather, we generated 8 variants of Mosquitto. We started with a binary that includes all features, which represent variant 0. This also gives us a baseline for #crashes present in the source prior to feature removal. We generated variants 1 to 7 in this way: variant i is obtained from variant $i-1$ by selecting and deactivating a feature that was active in variant i.

Finally, in principle, it is possible for features to be dependent on each other in such a way that removing a feature may break another one. Throughout the analyses of all the target programs, there have not been instances of multiple features utilizing or overlapping the same code blocks. They are outside of the core components of the project, and are fairly compartmentalized in that fully removing a given feature will not break the build. Should this situation arise, we expect that removing a feature without removing dependent features would result in breaking the build, which would still prevent an incorrect implementation from being generated.

Results. Review of code comparison reports did not provide evidence of any incorrectly identified or missing lines of code; we, therefore, conclude that our approach successfully identified **all code** related to the target features in Mosquitto.

Table 6 shows code coverage results obtained via fuzzing, including the time for which each fuzzing session was run. Diminishing execution times are due to the fact that removing a feature entails that related code paths no longer exist,

#Removed Features	Lines of Code			Functions			Total Time [min]
	Total	Covered	Coverage	Total	Covered	Coverage	
0	10087	6428	63.7%	377	319	84.6%	108
1	9666	6209	64.2%	370	314	84.9%	105
2	9232	5870	63.6%	339	289	85.3%	67
3	8819	5532	62.7%	332	281	84.6%	62
4	8079	4841	59.9%	326	261	80.1%	59
5	8013	4853	60.6%	324	260	80.2%	56
6	7112	4566	64.2%	296	251	84.8%	41
7	6927	4380	63.2%	292	247	84.6%	36

Table 6. Results of fuzz testing for 8 variants of Mosquitto

resulting in a shorter fuzzing session. For all but one variant, fuzzing covered **>60% of LOCs** and **>80% of functions**, **without identifying any bug** that was not originally present prior to feature removal. While dynamic testing does not *guarantee* functional correctness, or the complete absence of bugs, supplementing unit tests with fuzzing provides extra assurance that our feature removal did not inadvertently break another component. These results give us confidence that our approach results in functionally correct source code.

9 USER STUDY

An important experimental question concerns the usability of PRAT and the information it generates, including feature graphs. To evaluate this question, we carried an IRB-approved user study designed to compare how participants would interact with PRAT versus how they would go about the process of feature identification and removal manually. The goal of the study was to answer the following high-level research questions:

- **Question #1: Does the notion of "feature" as used by PRAT align with the intuitive notion of feature used by expert during code analysis?** While discrepancies exist, semi-structured interviews show that most participants used a working definition which directly matches the one used in PRAT's design.
- **Question #2: Does PRAT provide information useful to simplify code understanding?** Participants' impressions of the tool were largely positive and show that PRAT has potential to simplify the feature identification workflow, and provides useful information to human experts.
- **Question #3: Can PRAT improve human performance in feature identification tasks?** While we acknowledge that performance is expected to be highly dependent on expertise and the specific codebase under examination, results using PRAT show significant improvement in accuracy over manual code analysis.

9.1 Overall Study Design

The primary goal of the study is to understand whether software developers are likely to find PRAT useful and easy to use. As such, each participant in the study was asked to complete two tasks: (1) identifying code pertaining to a given codebase feature via manual analysis; and (2) completing the same task using PRAT. Tasks were conducted using a think-aloud protocol. Through observing the participant during the task, we could follow their intuition for defining what a feature is and determining where a given feature is implemented in a code base. Further, after completing the tasks, each participant was involved in a semi-structured interview in which we asked the following four questions:

- **Was there a discrepancy between your definition of a feature and PRAT's?** Defining what a feature is can vary from person to person. It is important that PRAT's definition of a feature is commonly held.

- **How did PRAT impact your workflow for feature identification and removal?** When evaluating the usability of PRAT, we need to guarantee that the benefits of using PRAT are worth learning a new toolchain.
- **Is PRAT's notion of a feature graph/web page output intuitive?** The artifacts that PRAT outputs are important for guiding a user through feature removal. Thus the reports need to be intuitive to users.
- **What are some pros and cons of using PRAT over manual identification and removal?** An open question for potential improvements to PRAT is useful to see if there are common improvements users would like to see.

During the experiments, we also measured human performance as number of lines of code correctly identified (discussed below).

9.2 Participant Recruiting

Initial participants were recruited via email and word-of-mouth advertisement among the CS graduate student population at Northeastern University and Worcester Polytechnic Institute; the rest were recruited through snowball sampling. We consider this user population (predominantly young graduate students) appropriate to test given our goals; as it ensures that participants are generally familiar with software development, but unlikely to have prior experience with the codebases under examination (which could skew results). Prior to the actual study, we conducted pilot testing with two subjects. For the actual experiment, we recruited 10 participants, consisting of 3 women and 7 men aged 24-35. One participant worked in industry while the remaining were graduate students. According to best practices, this sample size is appropriate for a semi-structured interview design like ours [33].

The recruiting materials indicated that the study would take 30-60 minutes and focus on the usability of a new tool used for program debloating. Participants were asked for consent to being recorded and compensated with a $15 gift card.

9.3 Study Protocol

At the beginning of each session we explained the task and the participant's role at a high level. We asked that the participants talk aloud to explain their thought process for each of the subsequent tasks. We presented the study to the participants as a task asking them to define what would constitute a feature agnostic of code base, and once they defined a feature, how would they go about finding all the implementation locations withing a given code base.

We then gave participants the Mosquitto [1] code base for experimenting with. They first started by proposing a definition of a feature as it relates to Mosquitto. Next, we asked them to explain how they would find all the implementation locations for one of the features they selected. Finally, we asked them to go about manually removing a given feature using the methodology they outlined.

After we gave participants the task of manually removing a feature from Mosquitto, we introduced them to PRAT. Once we explained PRAT to the participants, we asked them to use the tool to remove the same feature they had previously removed manually. Next, we asked the participants to examine the feature graph that PRAT generated, and explain whether the artifacts were intuitive or otherwise useful to the process. We visualized feature graphs as hierarchical HTML documents. The reports listed each files containing relevant lines; clicking on a filename would display a colorized representation of the lines located by PRAT.

While our experiments are within-subjects, we did not compensate for learning effects as we postulate that such effects do not arise. In our design, we asked each participant to carry manual analysis first, and then use PRAT. Because the set of lines identified by PRAT is independent of the user understanding of a given feature, the experience gained during manual

analysis is irrelevant to the output and performance of PRAT. Conversely, a different design involving using PRAT before manual analysis would risk incurring in such effects.

9.4 Data Collection

Due to the impact of the COVID pandemic, 7 sessions were conducted in person, and 3 were conducted remotely. The procedure was identical in both cases: during interviews, notes of user responses to our questions were taken; the subsequent analysis and results were generated based on these notes. We also recorded task completion time and number of lines correctly identified for each user and task.

9.5 Interview Results

We now present the results from our study's four questions.

Feature definition. Of the 10 participants, 3 defined the notion of a feature differently than the definition used by PRAT. One participant defined a feature as any component used by the codebase. This did not imply that the user had control over its configuration. Another participant said a feature was any formatting properties of a codebase such as ordering of fields in a packet. A third participant stated a feature to be the overview of an application. This would be the application's UI, name, or contents returned by running `man` against the target. The remaining 7 participants defined a feature in the same way as PRAT.

Impact on workflow. When asked how using PRAT would impact a participant's workflow, 6 participants responded by saying it speeds up the process of feature removal significantly. 3 other users did not explicitly comment on workflow speed-up, and 1 qualified that PRAT could speed up the workflow on large codebases, but smaller codebases could be analyzed more quickly by hand. 2 participants stated that PRAT is helpful for removal of useless code beyond what compiler optimizations can typically capture. According to 3 participants, PRAT may produce output that could be difficult for non-technical users to understand, and in instances of small codebases it may be easier to manually `grep` through the project to find code to remove. Another 2 participants stated that PRAT makes code understanding much simpler as the tool shows what the code to remove looks like, as opposed to just showing how many LOCs to remove. Seeing the context of the code block is a helpful aid for feature removal.

Feature graph and reports. All 10 of our participants responded that they thought the HTML-based reports PRAT generates are useful for guided feature removal. Of those, 3 participants stated that the feature graph can be unnecessary, especially if the user is unfamiliar with code. Another 4 of the participants said that the HTML reports give a simple, and effective overview of the targeted feature in the context of its source files. One participant preferred PRAT's `stdout` output, but followed by saying the HTML report is good for someone less familiar with the code as it is easier to understand. Three participants also suggested that the HTML reports be made to look a little more like code commit diffs on GitHub, since our reports are similar to that already.

Pros and cons. 3 participants responded that the automation is one of the biggest advantages of PRAT; with one participant specifically mentioning that an automated and complete workflow provided by PRAT makes the results more reliable than trusting a user's manual work. All of participants said that PRAT is fast and can save users a lot of time. But 2 participants also mentioned that large codebases may increase PRAT's running time because of multiple compilations PRAT needs when it analyzes programs in order to generate coverage diffs. Two of the participants stated that PRAT is easy to use, and

another 2 participants also suggested that it will be better if PRAT can provide a UI window for users to not only view results, but to interact with it during the removal process. Although one participant responded that they think PRAT's results are reliable, 3 participants said that PRAT still has the risk of introducing new bugs into programs, like breaking other features by removing incorrect code. Furthermore, the 3 users which provided a definition of feature different from PRAT's consistently stated that they would consider PRAT useful as a code understanding tool, but not as a debloating tool. One further qualified that they would prefer performing the actual debloating by hand, relying on their own skills.

9.6 Performance Results

We measured time for manual and PRAT-based feature identification; to avoid ambiguities, all times were rounded to the minute. However, we did not use these measures to quantify performance for two reasons: (i) due to the think-aloud protocol, it is likely that actual times would be lower in a non-lab setting; and (ii) performance of PRAT-based identification is dominated by compilation time and thus codebase-dependent. We report the measures for completeness: the mean manual tasks completion is 12 minutes (standard deviation is 4 minutes); all PRAT-based tasks completed in 2 minutes.

A more interesting question concerns accuracy, in terms of number of lines of code identified. On average, participants identified 2% of relevant lines of code (standard deviation 2%). All PRAT-based tasks identified all relevant lines of code, as verified in our correctness evaluation (ref. 8.7).

9.7 Take-Aways

Review of the semi-structured interviews evidences that PRAT's definition of feature aligns with most user's intuitive understanding. They also suggest that PRAT can simplify code removal and understanding and save developer's time, and feature graphs are a useful output format for the relevant information. However, a small number of participants also expressed concerns about clarity of the output, and the possibility of introducing bugs. This suggests that the most useful direction for future work may be providing further context, to increase user confidence in the correctness of PRAT's determinations.

In terms of performance, PRAT-aided removal appears to significantly improve human performance over manual analysis alone. One may argue that a user willing to invest time and practice into code understanding may eventually achieve the same result as PRAT; however, one of the benefits of PRAT is precisely to avoid the need for investing time in such activities.

Overall, we found the results of the study encouraging, as they show that PRAT has the potential to significantly simplify feature identification and removal.

10 DISCUSSION

Limitations of Dynamic Analysis and Testing. PRAT leverages dynamic analysis both for feature identification and for evaluating correctness. This comes with well-known limitations.

The extent to which PRAT can correctly identify unused code depends on the test set being sufficiently extensive and representative. In order to evaluate correctness, we performed extensive examination of the output of PRAT (ref. Section 8.7); results suggests that our test case generation strategy correctly identify feature-related code.

We also acknowledge that our approach to evaluating correctness has its own limits. Fuzzing and unit-testing alone cannot be taken as a *guarantee* of correctness, due to the intrinsic limitations of completeness of dynamic analysis. To

alleviate some of those concerns, we performed manual review of debloated code to flag instances of lines of code incorrectly associated to features, as discussed in Section 8.7. This analysis found no issues. While static analysis could in principle provide stronger guarantees than either dynamic or manual analysis, it comes with its own set of problems: relevant operations such as code reachability and points-to analysis are in general undecidable, and have in practice precision limitations [43], that limit the scope of the conclusions that can be extracted. Thus, we consider our current approach acceptable.

Integrating additional sources of information. Our approach shows that build system options are valuable in identifying non-essential features. For generality, PRAT assumes that these are the *only* sources of feature information, but in some cases, additional sources may be available. Well-maintained codebases frequently come with in-source documentation, such as Doxygen [5], which is both highly structured and explicitly associated with code entities (functions, variables, etc.). Parsing it using NLP techniques could enable the identification and removal of features at a finer level of granularity than based on build options alone. Another source of information is technical documentation associated with a software tool (e.g., standards, manuals, FAQs). While this data may not be sufficiently structured to support code analysis, it may contain information useful for understanding the purpose of different features. Identifying and integrating this information within feature graphs is an interesting future direction.

Generalizability. In order to demonstrate the applicability of PRAT beyond C/C++ codebases for which it was originally designed, we extended it to support Rust programs by using the Cargo build system and `kcov` code coverage tool. Furthermore, we included Quiche and rav1e, both written in Rust, in our evaluation set. PRAT's results on Quiche and rav1e (Sections 8.2 and 8.4) are comparable to those on the rest of the programs in our set, suggesting that the PRAT algorithm has general validity and results do not depend on the particular language in which a codebase is written.

11 THREATS TO VALIDITY

The main threat to *internal validity* is the lack of uniformity within build systems of projects. Since there is no definitive schema, the build systems tend to be free-form so long as they work. And when a large project has a commensurate number of contributors, the contents of the build system may become even more inconsistent (Section 4). Inconsistent/inaccurate build configurations could cause PRAT to silently fail to identify relevant features and/or lines of code in our experiments. To strengthen the internal validity, we reviewed the Mosquitto codebase in detail, to identify: (i) features missed by our approach; (ii) lines of code associated with relevant features, but not identified by PRAT; and (iii) lines of code wrongly associated with a feature. Our manual analysis did not find any such issue.

Another concern relates to *external validity*, given the variety of codebases within the scope of PRAT's goals. To mitigate issues of external validity, we applied our approach to a set of projects which differ in purpose, language, and build systems. Results suggest that PRAT adapts well to a diverse set of projects.

12 RELATED WORK

Program Debloating and Minimization. Quach et al. [38] present a debloating framework to identify and remove unused shared library code from memory. Rastogi et al. [48] use dynamic analysis to remove components in a Docker container that are not necessary to support the container's core application. These approaches are orthogonal to ours as they do not incorporate the notion of feature-driven code removal.

Another line of work focuses on the specific problem of minimizing failure-inducing test cases. Hierarchical delta-debugging [35], C-reduce [39], and Perses [47] are based on conceptual extensions to delta debugging [51]. They are generally designed to minimize a program while preserving a failure-triggering property, a goal that is orthogonal to ours.

Test Case-driven Debloating. CHISEL [28] aims to accelerate program reduction via online learning. The system works on source code and builds and refines a statistical model by repeated trial and error, which determines how much it is likely that each candidate program supports the desired functionality. RAZOR [37] performs code reduction for deployed binaries. Starting from test-cases, RAZOR uses several control-flow heuristics to infer complementary code that is necessary to support user-specified functionalities. Koo et al. [32] remove the shared libraries unnecessary for a user-given run-time configuration. Their system uses code coverage analysis to identify the association between run-time directives and libraries. Similar to Koo et al., TRIMMER [44] specializes an application to its deployment context by using user-provided configuration data. TOSS [22] starts from feature-related test cases and employs dynamic tracing and symbolic execution to identify feature-related code from the original program binary. These approaches differ from our system in that they require the user to provide expressive test cases precisely identifying the features of interest, while we use a higher-level, explicit notion of features automatically inferred from build configurations along with synthesized test cases for more comprehensive coverage.

Other Feature-Oriented Specialization Approaches. [29] proposes feature-based program customization based on static data flow analysis and program slicing to remove unnecessary components from Java bytecode. The user describes features of interests by specifying a set of *seed methods* within the callgraph. Also, CARVE [20] leverages fine-grained, developer-provided code annotations to bind LOCs to features, and uses a code analyzer to deactivated unwanted features. In contrast to these works, our system retrieves the features defined in the build configuration and maps the corresponding lines of code to the feature by performing a coverage diffing-driven approach. Therefore, it does not require the user to provide code-level feature specifications.

Feature Location and Variability Mining. There exists a significant body of work that focuses on locating software features within a codebase. As discussed in Section 1, much of these works rely on either the availability of test cases [26, 49] or externally provided knowledge [24, 30, 36, 40, 52], which make them unsuitable for our task. Also, these approaches tend to require that features are known *a priori*.

Test Case Synthesis. KLEE [21] is a symbolic execution tool that is capable of generating high-coverage test cases on C-based programs. This is the engine that we utilize for the generation of test cases due to how it functions with minimal user input. There have been other tools built on KLEE that aim to extend its functionality (e.g., KLOVER [34]).

In lieu of traditional design practices of static analysis, and extensive testing is *formal verification*. Formal verification of IoT devices is essential for the detection of vulnerabilities and guaranteeing functional correctness [31]. Model checking [23] can be used to ensure the accuracy of an implementation and show via proofs that it conforms to a given specification. However, this requires significant manual effort and extensive domain-specific knowledge of the system under test.

13 CONCLUSION

In this paper, we presented PRAT, a novel approach for the identification and removal of features from IoT-oriented third-party software components. PRAT automatically identifies available features by analyzing the build system and

visualizes features and their implementation details in a succinct form using the novel representation of *feature graphs*. Once the analyst has indicated which features they wish to remove, PRAT applies a form of debloating based on code coverage analysis. Based on the results, our approach is both fast and effective at removing all code related to features to be removed, without introducing new bugs. It also has potential to simplify developers' workflow by streamlining feature identification/removal compared to manual code analysis. By providing both decision support tools (feature graphs) and a practical debloating algorithm, we believe our proposed approach makes a significant contribution towards simplifying unwanted feature removal.

14 ARTIFACT RELEASE

We publicly released the code of our PRAT prototype on the OSF platform. The code can be accessed at https://osf.io/5n4zf/?view_only=0f1496d4918d499a9f5f9a05becb7aea.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their help with the revision of this paper. This project was supported by the Office of Naval Research (Grant#: N00014-18-1-2660). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agency.

REFERENCES

[1] 2018. Eclipse Mosquitto. https://mosquitto.org/
[2] 2019. AMQP library for C. https://github.com/Azure/azure-uamqp-c original-date: 2015-11-20T06:13:31Z.
[3] 2019. ccache - compiler cache. https://ccache.dev/
[4] 2019. darconeous/libnyoci: A flexible CoAP stack for embedded devices and computers. RFC7252 compatible. https://github.com/darconeous/libnyoci
[5] 2019. Doxygen: main page. http://www.doxygen.nl/
[6] 2019. GitHub - jtpereyda/boofuzz. https://github.com/jtpereyda/boofuzz
[7] 2019. Lora-net/LoRaMac-node: Reference implementation and documentation of a LoRa network node. https://github.com/Lora-net/LoRaMac-node
[8] 2019. obgm/libcoap: A CoAP (RFC 7252) implementation in C. https://github.com/obgm/libcoap
[9] 2019. OpenDDS. https://opendds.org/
[10] 2019. zeromq/libzmq: ZeroMQ core engine in C++, implements ZMTP/3.1. https://github.com/zeromq/libzmq
[11] 2020. aom - Git at Google. https://aomedia.googlesource.com/aom/
[12] 2020. cloudflare / quiche: Savoury implementation of the QUIC transport protocol and HTTP/3. https://github.com/cloudflare/quiche
[13] 2020. CVE - Common Vulnerabilities and Exposures. https://cve.mitre.org/
[14] 2020. FFmpeg. https://ffmpeg.org/
[15] 2020. xiph / rav1e: The fastest and safest AV1 encoder. https://github.com/xiph/rav1e
[16] 2021. AFLNet: A Greybox Fuzzer for Network Protocols. https://github.com/aflnet/aflnet
[17] 2021. american fuzzy lop - a security-oriented fuzzer. https://github.com/google/AFL
[18] David Adrian, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, Paul Zimmermann, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, and Emmanuel Thomé. 2015. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*. ACM Press, Denver, Colorado, USA, 5–17.
[19] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cedric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. 2015. A Messy State of the Union: Taming the Composite State Machines of TLS. In *2015 IEEE Symposium on Security and Privacy*. IEEE, San Jose, CA, 535–552.
[20] Michael D Brown and Santosh Pande. 2019. CARVE: Practical Security-Focused Software Debloating Using Simple Feature Set Mappings. *CoRR* abs/1907.02180 (2019), 8.
[21] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, Berkeley, CA, USA, 209–224.

[22] Yurong Chen, Shaowen Sun, Tian Lan, and Guru Venkataramani. 2018. TOSS: Tailoring Online Server Systems Through Binary Feature Customization. In *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation* (Toronto, Canada) *(FEAST '18)*. ACM, New York, NY, USA, 1–7.

[23] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. 1999. *Model Checking*. MIT Press, Cambridge, MA, USA.

[24] Marc Eaddy, Alfred V Aho, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. 2008. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *2008 16th IEEE International Conference on Program Comprehension*. Ieee, 53–62.

[25] Lars Eggert. 2020. Towards Securing the Internet of Things with QUIC. In *Workshop on Decentralized IoT Systems and Security (DISS)*.

[26] Andrew David Eisenberg and Kris De Volder. 2005. Dynamic feature traces: Finding features in unfamiliar code. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, 337–346.

[27] Jonathan Foote. 2015. How to Fuzz a Server with American Fuzzy Lop. https://www.fastly.com/blog/how-fuzz-server-american-fuzzy-lop.

[28] 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* (Toronto, Canada) *(CCS '18)*. ACM, New York, NY, USA, 380–394.

[29] Y. Jiang, C. Zhang, D. Wu, and P. Liu. 2016. Feature-Based Software Customization: Preliminary Analysis, Formalization, and Methods. In *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*. 122–131.

[30] Christian Kästner, Alexander Dreiling, and Klaus Ostermann. 2013. Variability mining: Consistent semi-automatic detection of product-line features. *IEEE Transactions on Software Engineering* 40, 1 (2013), 67–82.

[31] K. Keerthi, Indrani Roy, Aritra Hazra, and Chester Rebeiro. 2019. *Formal Verification for Security in IoT Devices*. Springer International Publishing, Cham, 179–200.

[32] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. 2019. Configuration-Driven Software Debloating. In *Proceedings of the 12th European Workshop on Systems Security - EuroSec '19*. ACM Press, Dresden, Germany, 1–6.

[33] Anton J Kuzel. 1992. Sampling in qualitative inquiry. (1992).

[34] Guodong Li, Indradeep Ghosh, and Sreeranga P. Rajan. 2011. KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 609–615.

[35] Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*. 142–151.

[36] Denys Poshyvanyk, Yann-Gael Gueheneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. 2007. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering* 33, 6 (2007), 420–432.

[37] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1733–1750.

[38] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. In *USENIX Security*. 869–886.

[39] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 335–346.

[40] Meghan Revelle, Bogdan Dit, and Denys Poshyvanyk. 2010. Using data fusion and web mining to support feature location in software. In *2010 IEEE 18th International Conference on Program Comprehension*. IEEE, 14–23.

[41] Steve Schmidt. 2015. Introducing s2n, a New Open Source TLS Implementation. https://aws.amazon.com/blogs/security/introducing-s2n-a-new-open-source-tls-implementation/

[42] 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)*. ACM, New York, NY, USA, 552–561.

[43] Marc Shapiro and Susan Horwitz. 1997. The Effects of the Precision of Pointer Analysis. In *Static Analysis*, Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Pascal Van Hentenryck (Eds.). Vol. 1302. Springer Berlin Heidelberg, Berlin, Heidelberg, 16–34.

[44] 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* (Montpellier, France) *(ASE 2018)*. ACM, New York, NY, USA, 329–339.

[45] Anna Kornfeld Simpson, Franziska Roesner, and Tadayoshi Kohno. 2017. Securing vulnerable home IoT devices with an in-hub security manager. In *PerCom Workshops)*. IEEE, Kona, HI, 551–556.

[46] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *NDSS*, Vol. 16. 1–16.

[47] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering*. 361–371.

[48] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. 2017. Cimplifier: Automatically Debloating Containers. In *ESEC/FSE*.

[49] Norman Wilde and Michael C Scully. 1995. Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance: Research and Practice* 7, 1 (1995), 49–62.

[50] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Security Symposium (Security)*. Baltimore, MD.

[51] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.* 28, 2 (Feb. 2002), 183–200.

[52] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang. 2006. SNIAFL: Towards a static noninteractive approach to feature location. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 15, 2 (2006), 195–226.