An Initial Investigation of Protocol Customization

David Ke Hong
University of Michigan
kehong@umich.edu

Qi Alfred Chen
University of Michigan
alfchen@umich.edu

Z. Morley Mao
University of Michigan
zmao@umich.edu

ABSTRACT

Attacks exploiting design or implementationfl aws of particular features in popular protocols are becoming prevalent and have led to severe security impacts on a majority of software systems. Protocol customization as a general approach to specialize a standard protocol holds significant promise in reducing such attack surfaces in common protocols. In this work, we perform an initial investigation of applying protocol customization practices to reduce the attack surface of standard protocols. Our characterization study on 20 medium or high-impact common vulnerability exposures (CVEs) published in recent years indicates that some forms of customization have been supported in existing protocol software, but were implemented with huge manual effort and in an ad-hoc manner. More systematic and automated ways of protocol customization are awaited to generalize common customization practices across protocols. To work towards this goal, we identify key research challenges for the support of systematic and sufficiently automated protocol customization through real-world case study on popular protocol software, and propose an access control framework as a principled solution to unify existing protocol customization practices. We also present a preliminary design of a protocol customization system based on this design principle. Preliminary evaluation results demonstrate that our proposed system supports common customization practices for a majority of real-world protocol vulnerabilities in a systematic way.

1 INTRODUCTION

Recent years have seen severe attacks targeting core protocols that many large-scale production systems as well as billions of mobile and IoT devices heavily depend on, leading to massive information stealing, privacy leaks and notorious ransomeware [4, 36, 37, 39]. With attack surfaces rooted in the design or implementation of protocols, patching the vulnerability becomes challenging to developers as it requires comprehensive and deep understanding of latest protocol details. Moreover, upgrading with the patched protocol software is hard for production systems, as many of them have stringent requirements on high availability and cannot afford interruption time for protocol upgrading. Though both the industry and research community have put in significant endeavor for hardening the security of various network and application protocols [27, 31], protocol vulnerability threats are yet to be prevented.

FEAST'17, November 3, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-5395-3/17/11…$15.00
https://doi.org/10.1145/3141235.3141236

We observe that a key enabler of many such attacks lies in some exploitablefl aw in the design or implementation of a particular feature or extension of a protocol. Due to the consideration of general support, many popular feature-rich protocols, such as SSL/TLS, HTTP and SSH, are implemented as a one-size-fits-all library or software package. In common deployment scenarios, many of the features become extraneous and may have never been invoked by applications, but are invokable by attackers and represent an attack surface that could have been prevented. For example, the infamous HeartBleed vulnerability results from an implementation flaw with the TLS/DTLS HeartBeat extension in OpenSSL. Though many applications do not require the heartbeat feature to run the SSL protocol, it is implemented and enabled as a standard feature in OpenSSL. To address such embarrassingly risky situations, effective and systematic solutions are awaited to provide accurate understanding of feature requirements andfl exible support of feature disabling.

Protocol customization is a set of techniques to modify or transform a standard protocol and generate a specialized protocol which only supports a list of required features (analogous to feature whitelisting), among which subsetting is a technique that removes all the features except required ones from the protocol implementation, and dialecting is a technique that modifies the core functionality of the standard protocol with the resulting protocol partially compatible to the standard protocol. Protocol customization holds significant promise in proactively eliminating potential vulnerabilities associated with unused protocol functionality. For example, the latest ransomware vulnerability is mitigated by subsetting the unsecured, 30-year-old SMBv1fi le-sharing protocol on your Windows systems and servers [32]. However, existing protocol customization practices are conducted in an ad-hoc manner by developers and far from systematic. Furthermore, customization supports in existing protocol software are manually implemented and thus prone to human errors. Wefi nd that many popular protocol software or library packages have provided some forms of customization, but are usually at too coarse granularities to reach a sweet spot of security and usability.

To address the aforementioned inefficacies, we conduct an initial investigation of applying protocol customization to reduce the attack surface of standard protocols. Firstly, we characterize recent CVE reports and patches with common protocols, and identify that protocol customization is an effective solution to common protocol vulnerabilities today. Secondly, we identify key research challenges in realizing a systematic and sufficiently automated customization process for a wide range of protocols. Finally, we propose one promising solution direction, feature access control, as a unifying solution to support a variety of protocol customization practices, and propose a preliminary system design to support it. Preliminary case studies with real-world protocol vulnerabilities and software

validate our identified challenges and solution direction. Our exploration generates real-world insights and raises initial directions for future work in this space.

2 CVE STUDY

In order to understand how protocol customization can effectively address realistic threats such as the well-known HeartBleed vulnerability, we study 20 medium or high-impact CVEs published in the last 10 years targeting both system-level network protocols (e.g., HTTP, SSL/TLS) and application-layer protocols (e.g., OpenSSH, XMPP). Wefi nd that all vulnerabilities stem from the implementation or designfl aw of some protocol feature or extension, most of which are not commonly used but enabled by default. Moreover, a majority of them can be mitigated with some forms of feature customization support from the protocol implementation, such as compile-time or runtime module disabling or parameter tuning. We categorize the types of protocol customization approach for each CVE in Table 1, where adding protocol extension can be considered as a form of dialecting, while parameter tuning and adding bounded conditions are forms of subsetting. This characterization study guides us to believe that protocol customization is promising to serve as a general solution to reduce attack surface and mitigate vulnerabilities for a broad range of protocols.

Customization approach	CVEs
Compile-time disabling	Heartbleed [14], RC4 [16], FREAK [15]
Runtime disabling	HTTP_PROXY redirection [23], Apache integer overflow [9], XMPP dialback [20], XMPP message carbons [25], OpenSSH information leak [19], HTTP proxy data leak [6], Apache XSS [10], FREAK [15], Logjam [17], RC4 [16], CRIME [11], BREACH [13]
Runtime parameter tuning	Slow read DoS [22], Apache integer overflow [9], Range header DoS [8]
Patching with bounded conditions	Dependency cycle DoS [18], HPACK bomb [21, 24], HTTP proxy DoS [5]
Adding protocol extension	TLS renegotiation [7], Lucky13 [12]

Table 1: Categorization of protocol customization approach (some CVEs with multiple forms of customization)

3 RESEARCH CHALLENGES

As informed by our CVE study, some protocol software packages have already supported certain forms of protocol customization leveraging compile time options or run-time configurations. However, currently these customization practices are provided by manual and developer-specific code base instrumentation, and thus are limited in multiple aspects. First of all, such approach is difficult to comprehensively cover all important customization options. For example, the HPACK bomb vulnerability [21, 24] can befi xed by limiting the size of header, but the developer failed to cover such customization option at the time when the vulnerability is discovered. Second, since the required code base instrumentation is largely manual, the current customization technique is error-prone. Third, due to the extra efforts involved, it is difficult for developers to provide sufficientlyfi ne-grained customization options for maintaining protocol usability. For instance, it is relatively easy for developers to support the customization options that disable the entire *mod_proxy* module, but for applications relying on this module for their core functionality, customization options at such coarse granularity are not useful at all. Customization atfiner granularity, e.g., submodules inside those module, are more useful, but can be practically impossible due to the largely increased manual efforts. Defining the right granularity to support feature customization itself is a challenging problem. These problems call for a more general, systematic, and sufficiently automated approach for identifying and enforcing customization options in the protocol customization process.

3.1 Customization Options Identification

To support protocol customization in a target protocol, the necessaryfi rst step is to identify the set of customization options that are of interest for attack surface reduction. These options are provided to the system administrators to help more efficiently make customization decisions. Each customization option needs to specify two parts of information: (1) which protocol feature to customize, and (2) what customization method to use. Due to the diversity in protocol types and design and implementation choices, requiring administrators to manually identify these options can hardly be applied in practice. Thus, the research question is how to provide a more systematic and automated approach to identify potential customization options for a given protocol.

Effectively addressing this problem involves the following research challenges.

(1) **Systematically understand the attack surface**: Before identifying customization options for a given protocol to mitigate security risks, it is necessary to have a systematic understanding of its attack surface. For example, to select features for customization, we need to define which features are likely to expose vulnerabilities when deployed. This is essentially a prediction task based on the knowledge of potentially vulnerable protocol design and implementation choices, which typically requires years of experience in manual protocol security analysis, and thus makes it a challenging task to develop a systematic and automated approach.

(2) **Identify features that are semantically meaningful and self-contained**: The customization options provided to the administrators need to be semantically meaningful, e.g., at a reasonably high level with well connection to the human-readable protocol descriptions, so that they are easy to understand and control. Meanwhile, since these customization options will be executed in the actual implementations of the protocol, the features to be customized need to be also relatively self-contained at the implementation level so that they can be conveniently mapped to the code-level entities, e.g., functions and modules, to perform customization. This requires a systematic approach to connect high-level protocol features to low-level implementation details.

3.2 Customization Options Enforcement

With some customization options identified, the next step is to execute the customization requests. Thus, the research question is how to design a systematic approach to support protocol customization for a given protocol and its identified customization option. As discussed at the beginning of this section, such approach needs to be sufficiently automated to increase efficiency and prevent human errors. To solve this problem, we need to address the following research challenges.

(1) **Systematically locate protocol feature code for customization**: For the vulnerabilities collected in our CVE study in §2, the customization suggestions are identified manually leveraging developers' understanding of the code base structure related to the protocol features. To automate this process, a novel approach is required to efficiently analyze the code base and effectively localize code-level implementation of each feature to be customized (a.k.a, *feature code*).

(2) **Effectively support protocol customization with minimized manual efforts**: After locating the feature code, customization support needs to be correctly added and also be controlled by the user, e.g., via compile-time options or runtime configurations. Existing customization support heavily depends on the developer's initial design of code base structures. For example, to provide compile-time customization in Apache HTTP server, the source code of each module is located in a directory under "modules", and after applying `configure`, each configured module has a separate Makefile auto-generated inside its subdirectory to be included in the compilation [1]. However, since we aim to automate the process with minimized manual efforts from developers, it remains a challenge how to perform customization without assuming that the code base structure is ready for customization by design. When only the protocol binary is available, this is even more challenging since binary analysis is inherently harder than source code analysis due to the lack of high-level, semantically rich information about data structures and control constructs.

(3) **Support diverse types of protocol customization in the design**: The challenges in performing customization are further escalated when systematizing the support for different types of customization. For example, there are two types of customization, disabling modules and tuning parameters (e.g., window sizes). To realize the former, certain feature code needs to be excluded at runtime, while one solution to the latter still keeps the feature code but uses a set of parameters to form predicates to restrict its execution. These separate solutions need to be coherently integrated into the overall system design.

4 SOLUTION DIRECTION

To handle various protocol or application-specific constraints, we propose an access control framework (illustrated in Figure 1) to unify existing protocol customization practices. We call our application of access control to protocol customization as *protocol feature access control*. Protocol features are the resources to control, and the use of these features, e.g., unconditionally or conditionally

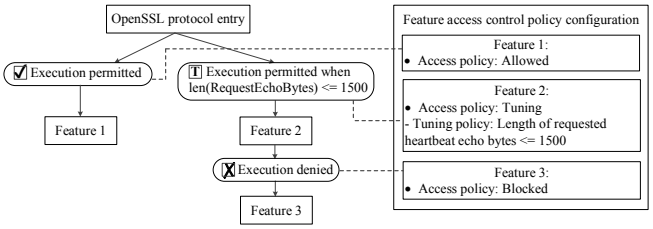

Figure 1: Protocol feature access control framework. "T" means the feature access control policy contains tuning parameters.

blocking a feature, are specified and controlled along the execution path by feature access control policies. As a classic area in computer security, access control has a set of relatively mature design principles, models, and best practices [58, 85, 86, 101]. Thus, utilizing access control design theories and techniques is a principled, general and systematic approach to support protocol customization. Furthermore, we propose to leverage recent advances in program analysis and machine learning (detailed as follows) to make protocol customization sufficiently automated.

4.1 Program Analysis based Approach

Program analysis enables automatically analyzing the behavior of computer programs regarding a program property of interest, e.g., performance [40], correctness [72], and security [52]. With the main benefit in automation, program analysis has been proven a powerful tool for systematically understanding and solving a wide range of computer software problems [41, 63, 67, 93, 96, 100, 102]. Building upon previous successes, we envision several program analysis techniques to be applied to address following challenges for realizing a practical system with systematic and sufficiently automated support of protocol customization.

(1) **Extract clues for protocol customization**: In order to systematically support protocol customization, it is necessary tofi rst understand what types of vulnerabilities can be mitigated by protocol customization, and what useful types of customization methods are. These can be effectively learned from various sources of vulnerability data, e.g., code snippets in the common vulnerability exposure (CVE) reports and patches (if available), but due to the large data volume, it is not feasible to manually read all of these code snippets. To address this challenge, we can leverage control and datafl ow analysis to automatically extract vulnerability code patterns in the reports and customization methods in the patches. These data are then fed into an analytic system (detailed in §5.1) that harnesses machine learning techniques to automatically recommend customization options to administrators.

(2) **Automate protocol customization**: To conduct protocol customization, the proposed system needs tofi rst analyze the code for places to customize, and then transform the code to enforce the customization request. In this process, the major challenge is to make these two steps sufficiently automated to increase efficiency and prevent human errors. To address

these challenges, we can leverage program analysis such as taint analysis techniques, with the help of machine learning techniques if needed, for thefi rst step to automatically identify and analyze the program properties to localize where to customize. For the second step, we can leverage program transformation techniques [70, 77], a typical component in program analysis tools, to modify the code to support protocol customization.

(3) **Provide functional correctness guarantees**: After a protocol is customized, it is necessary to ensure that the allowed protocol functionality can still work correctly. Previous works that apply program analysis to analyze program correctness (e.g., algorithm correctness [72], protocol correctness in both design and implementation [45, 46, 79]) inspire us to leverage data and control dependency analysis techniques for feature dependency analysis, which ensures the allowed protocol features are not affected by the customization.

(4) **Providefl exible customization options**: Protocol customization is not simply feature removal. Wefi nd that there exist protocol customization practices that still allow a protocol feature to function but under limited conditions. For example, the suggestedfi x for the HPACK bomb vulnerability [21, 24] is to limit the header size instead of completely disabling the HTTP/2 support. We call this type of customization as *parameter tuning*. This is a more practical solution, since unconditionally disabling a feature may not be accepted if the feature is a core functionality. However, how to configure the tuning parameter is a challenging problem since the tunable parameters may not be fully documented. If configuration is inappropriately set, it may impact the functional correctness. To address this challenge, we can leverage symbolic execution techniques [83] to systematically identify the set of parameters and their values for the legitimate use of a feature (detailed in §5.2).

(5) **Support customization for protocol executables**: Program analysis techniques typically target source code, which is not always available, e.g., when analyzing commercial products. Techniques that directly analyze binaries [89, 93] are attractive for security analysis targeting COTS (common off-the-shelf) or malicious programs. We can build upon these binary analysis systems to provide customization support even when only the protocol executables are available.

4.2 Machine Learning based Approach

Building upon the history of applying machine learning techniques to various security domains, such as malware detection [42, 80, 103], program analysis [43], intrusion detection [71, 92], policy enforcement [97], we see following opportunities in leveraging recent advances in machine learning to automate the protocol customization pipeline.

(1) **Determine protocol features to be customized**: as discussed in § 4.1, vulnerability patterns can be characterized from analyzing various sources of vulnerability data using natural language processing (NLP) techniques augmented with deep learning models [60, 91]. Semantic interpretation techniques [75] in NLP are demonstrated useful to extract structured semantic information by automated learning over corpora of natural language examples. The recent emergence of NLP systems [3, 29] into production further demonstrates the practicability of such techniques in understanding unstructured human texts. By leveraging the advancements in NLP, we propose to extract high-level features from protocol specifications and correlate them to the vulnerability patterns for identifying features to be customized (detailed in §5.1).

(2) **Enforce customization strategies**: In the feature localization step, one major challenge is bridging the gap between the interpretation of a protocol feature in some human-understandable form and at the code implementation layer. Previous works show great promise in using NLP techniques to map user expected software behaviors to corresponding functions in the software implementation [78, 81]. We thus propose to leverage NLP to learn semantically meaningful features from protocol specifications and documents, which are then mapped to the relevant code pieces using supervised and unsupervised learning methods.

(3) **Feature usage monitoring**: feature usage logs from runtime monitoring can be used to suggest refinement of current customization configuration. Reinforcement learning (RL), when combined with deep neural networks (a.k.a., deep RL), has shown unique advantages in adapting to real-world dynamics for decision making in network systems [26, 47, 76]. We propose to leverage RL to adjust tuning parameters of a customization configuration based on usage logs.

(4) **Transfer customization knowledge to new protocols**: when new types of protocols are incorporated into the customization environment, before observing enough vulnerability exposures, the invariants of customization knowledge based on existing protocols can be identified by transfer learning techniques as deep neural network based representations [44], which are then used to predict vulnerable features and recommend proper customization strategies for a new protocol.

5 PRELIMINARY DESIGN

In this section, we present a preliminary design of an analytic system for recommending customization options. Given the features to be customized, we further design a feature access control system to specify access control policies for each of them and realize the policy enforcement at runtime.

5.1 Customization Option Analytic System

To identify customization options for constructing feature access control policies, we propose an analytic system (illustrated in Figure 2) that takes various sources of protocol and vulnerability-related data, such as protocol specification, protocol software and CVE database, and leverages aforementioned program analysis and machine learning techniques to automatically recognize a set of high-level features and correlate them to CVEs (if any) and code-level implementation. The features with strong correlation to CVEs are identified as vulnerable features and recommended to system

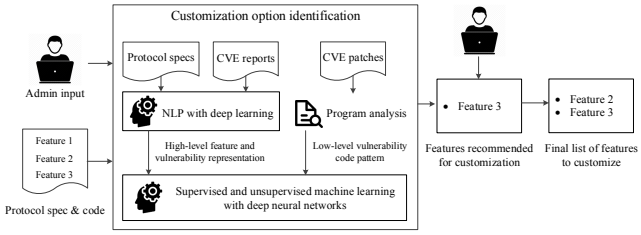

Figure 2: An analytic system to recommend customization options to system administrators for a given protocol.

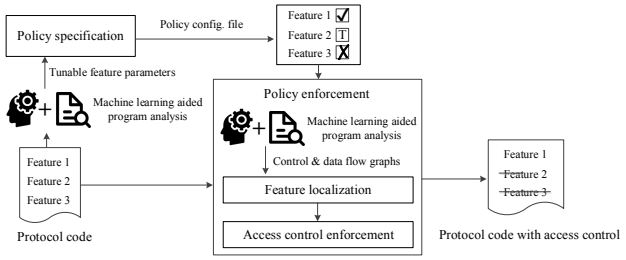

Figure 3: Feature access control system design. "T" means that feature is customized by tuning parameters, e.g., the policy configuration for Feature 2 in Figure 1.

administrators as customization options, while the features to be customized arefi nalized by administrators.

5.2 Feature Access Control System

Given the features to be customized (e.g., output of Figure 2) as input, a feature access control system, shown in Figure 3, generates corresponding access control policies and enforces them in the protocol implementation. Following the classic design of access control systems, our proposed design has two components: policy specification, and policy enforcement. In the specification component, the user describes the protocol customization requirements in the form of a list of feature access control policies. To support runtime customization, such specifications are included in a policy configurationfi le so that users can make customize changes dynamically through reconfiguring thisfi le.

Policy specification. To support the popular protocol customization practices today, we propose to define two types of policies (illustrated in Figure 1): *feature disabling* policies, and *feature tuning* policies that manifest the conditions of legitimate feature usage by limiting the range of tunable parameters for a targeted feature (a.k.a., *tuning parameters*). This design can be extended when there are future needs, which only needs updates in the underlying components without changing the whole system design. Such extensibility andfl exibility in supporting multiple forms of customization is an advantage by design due to the use of access control methodology in solving the protocol customization problem. In the support of feature tuning policies, the set of tuning parameters may not be fully documented. Thus, both identifying the set of tuning parameters and determining the legitimate value of each parameter are challenging tasks, and are important to ensure the usability and effectiveness of the system. For the latter, it may also affect correctness since setting inappropriate parameters may cause malfunction of the protocol feature. To address this challenge in a systematic way, we propose to use control and datafl ow analysis to identify the parameters that can affect the feature functionality for forming a candidate set of tuning parameters, and further apply symbolic execution to narrow down to the parameters (and their legitimate values) that are relevant to a given feature.

Policy enforcement. For the policies specified by the administrators, the next design component in our system is policy enforcement at runtime. As discussed in § 3.2, the key technical challenge is to systematically and effectively support protocol customization with minimized manual efforts. To address this challenge, we propose to use program analysis techniques, which are capable of automatically identifying and analyzing the program properties of interest from the program code. These techniques can be applied to both protocol source code and binaries. Our proposed solution has two steps: (1) feature localization, and (2) access control enforcement. The localization step searches the protocol implementation to identify the parts of code related to the targeted feature in the customization policy. The access control enforcement step then leverages the structure or properties of the identified feature code to apply customization. In the subsequent step, access control enforcement, if it is convenient to directly modify the protocol code, e.g., when the source code is given, we propose to directly add access control logic around the feature code chunk. For example, to disable the feature, access control policy checks are added at the code entries to prevent the feature code from being executed. Compared to other popular access control enforcement mechanisms, e.g., adding runtime policy enforcement layer, this approach incurs much smaller performance overhead since the access control is directly embedded in and compiled with the code base.

6 CASE STUDY

In this section, we present case studies on several popular network protocol implementations and code patches for the CVEs in § 2. Preliminaryfi ndings validate the need of automation support for protocol customization to address real-world challenges and the feature access control framework as a general solution to unify common customization practices for different protocols.

OpenSSL. Through analyzing the source code of OpenSSL (v1.0.1f), wefi nd at least 97 compilerfl ags (with prefix *OPENSSL_NO_*) that have been manually defined by developers for disabling various cipher suites, protocol extensions and other features. Moreover, these flags appear around 2460 times in 415 sourcefi les across the whole code base. In particular, the library consists of 24 code pieces across 15 sourcefi les for implementing the TLS/DTLS Heartbeat extension, which are wrapped by the compilerflag *OPENSSL_NO_HEARTBEATS*. On one hand, the existence of many compilerfl ags in OpenSSL show that feature customization through segregating and disabling feature-related code pieces may be realistic for real-world protocol software. On the other hand, it remains a challenging and error-prone task for developers to manually implement new customization options in the existing code base, especially when it is

expanding to incorporate new features or extensions. The significant manual efforts required and potential human errors involved can be minimized if automated feature localization and customization enforcement are supported.

Apache HTTP Server. Key HTTP/2 features, includingfl ow control, stream dependency & priority, stream multiplexing and compression, are not demarcated using compile-time or runtimeflags in the HTTP/2 module of the latest release of Apache HTTP Server (v2.4.x). These features are either all enabled or all blocked (i.e., by tuning the configuration parameter *mod_http2*) at runtime. As a consequence, it is impossible to selectively enable some features and disable the rest for avoiding vulnerabilities caused by features (e.g., compression, dependency & priority) to be disabled, while ensuring HTTP/2 continues to operate. Our source code study shows that the code structure of the HTTP/2 module is based on the HTTP/2 primitives and constructs (e.g., stream, session, worker, multiplexer) in an object-oriented fashion, upon which the new features are implemented. The implementation of different features are tightly coupled across the sourcefi les and can hardly be manually sliced for realizing customization at the feature-level granularity. Though 18 directives exist to allow runtime configuration of HTTP/2, they are insufficient to be used to mitigate vulnerabilities caused by certain HTTP/2 features [18, 21, 24], because developers hardly have any clues on possible attack surfaces before security experts discover them. Such challenge guides us to believe that automated feature customization, with feature-related code pieces localized and segregated in advance, is critical for responsive defense against zero-day attacks targeting newly discovered vulnerable features.

Dependency cycle DoS. The stream dependency & priority feature in HTTP/2 enablesfl exible multiplexing of concurrent streams in one connection, but can lead to severe vulnerability [18]. First, the RFC recommends the size of the dependency streams to be at least 100 for parallelism consideration but does not limit its upper bound [28]. Thus, a malicious client may fool a server to create many concurrent streams for consuming its memory. Second, corner cases (e.g., dependency cycles, rapid changes of dependencies), if ignored, may result in heavy CPU or memory consumption. We review the commit history of nghttp2 [30] andfi nd that several patches have been made for refining the legitimate check for this feature over the past few years [2, 33, 34, 38], including one for defining a sensible upper bound and a later one for adding a self-dependency check for detecting cycles. One important observation from this case is that identifying a proper set of tuning parameters and their legitimate values in real world is both challenging and critical. Inappropriate parameters (or values) may cause the attack mitigation useless or even malfunction of a useful feature.

Access control model validation. We investigate code patches for the CVEs in § 2 and discover that in 8 out of 20 cases extra condition check is added to restrict the legitimate circumstances for allowing the vulnerable feature to be invoked, while the majority of the rest arefi xed by unconditionally disabling it. As shown in Table 2, 17 of them can be expressed by feature disabling policies or feature tuning policies with appropriate tuning parameters. This study shows that the feature access control system is a promising direction to unify common customization practices for various protocols.

Policy type	CVEs
Feature disabling	XMPP dialback [20], XMPP message carbons [25], OpenSSH information leak [19], HTTP proxy data leak [6], FREAK [15], Logjam [17], RC4 [16], CRIME [11], BREACH [13]
Feature tuning	Heartbleed [14], Dependency cycle DoS [18], Slow read DoS [22], HPACK bomb [21, 24], Apache integer overflow [9], HTTP proxy DoS [5], HTTP_PROXY redirection [23], Range header DoS [8]

Table 2: Feature access control policy to CVE cases

7 RELATED WORK

We identify several research areas that are closely related to protocol customization. In the line of program analysis, various static and dynamic analysis techniques [48, 49, 54–56, 59, 61, 68, 73, 84, 90, 94, 95] are proposed to make control and datafl ow analysis on source code or binary increasingly effective. More closely to protocol customization, program analysis techniques are applied to analyze protocol security and correctness [45, 46, 50, 51, 69]. Besides, static or dynamic binary rewriting techniques are proposed to transform a program to meet some customization goals (e.g., bloatware mitigation) [57, 62, 65, 66, 74, 98, 99]. As future work, we plan to extend existing program analysis and customization techniques for realizing a practical system to support our protocol customization solution. In the line of machine learning, deep learning techniques are recently leveraged in binary analysis [53, 88], intrusion detection [64] and malware analysis [87]. Our work extends the deep learning approach to new security domains: protocol vulnerability identification and prevention. Our proposed protocol customization system involves a number of compute-intensive program analysis and machine learning tasks. To address the system challenge of meeting performance requirements, we plan to perform some study on the performance overhead and bottleneck of our proposed system, and explore the possibility of leveraging previous works [35, 82] that harness the cluster computing power to parallelize program analysis and machine learning tasks for signifcant performance speedup.

8 CONCLUSION

In this work, we highlight key research challenges of mitigating security vulnerabilities through protocol customization and propose a preliminary approach to address each of them. Preliminaryfindings from real-world case studies show that our proposed feature access control framework is a promising solution to unify common protocol customization practices.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their helpful feedback. We also thank Dr. Sukarno Mertoguno for his critical feedback on our work.

REFERENCES

[1] Apache HTTP Server configure - Configure the source tree. http://httpd.apache.org/docs/2.4/programs/configure.html.
[2] Check request/response submission error based side of session. https://github.com/nghttp2/nghttp2/commit/bb6f842b37b57c3d8e191db948e9165c59af7daf.

[3] Cloud Natural Language API. https://cloud.google.com/natural-language/.
[4] Customer Guidance for WannaCrypt Attacks. https://blogs.technet.microsoft.com/msrc/2017/05/12/customer-guidance-for-wannacrypt-attacks/.
[5] CVE-2008-2364: mod_proxy_http DoS. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-2364.
[6] CVE-2009-1191: mod_proxy_ajp data leak. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1191.
[7] CVE-2009-3555: SSL/TLS renegotiation attack. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-3555.
[8] CVE-2011-3192: Range header DoS. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-3192.
[9] CVE-2011-3607: Integer overflow in Apache HTTP server. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-3607.
[10] CVE-2012-3499: Apache XSS. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-3499.
[11] CVE-2012-4929: CRIME attack. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-4929.
[12] CVE-2013-0169: Lucky13 attack. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0169.
[13] CVE-2013-3587: BREACH attack. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-3587.
[14] CVE-2014-0160: Heartbleed bug. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160.
[15] CVE-2015-0204: OpenSSL FREAK attack. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0204.
[16] CVE-2015-2808: RC4 attack. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-2808.
[17] CVE-2015-4000: Logjam attack. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-4000.
[18] CVE-2015-8659: Dependency cycle DoS. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8659.
[19] CVE-2016-0777: OpenSSH client information leak. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0777.
[20] CVE-2016-1232: Prosody XMPP dialback vulnerability. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-1232.
[21] CVE-2016-1544: HPACK bomb. https://nghttp2.org/blog/2016/02/11/nghttp2-v1-7-1/.
[22] CVE-2016-1546: Slow read DoS. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-1546.
[23] CVE-2016-5387: HTTP_PROXY redirection. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5387.
[24] CVE-2016-6581: HPACK bomb. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-6581.
[25] CVE-2017-5858: XMPP Message Carbons extension vulnerability. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5858.
[26] Deepmind ai reduces google data centre cooling bill by 40 https://deepmind.com/blog/deepmind-ai-reduces-google-data-centre-cooling-bill-40/.
[27] Google Application Security. https://www.google.com/about/appsecurity/research/.
[28] Hypertext Transfer Protocol Version 2 (HTTP/2). https://http2.github.io/http2-spec/.
[29] Introducing DeepText: Facebook's text understanding engine. https://code.facebook.com/posts/181565595577955/introducing-deeptext-facebook-s-text-understanding-engine/.
[30] nghttp2 - HTTP/2 C Library and tools. https://github.com/nghttp2/nghttp2/.
[31] OpenSCAP. https://www.open-scap.org/.
[32] Petya Ransomware Spreading Rapidly Worldwide, Just Like WannaCry. http://thehackernews.com/2017/06/petya-ransomware-attack.html.
[33] Return error from nghttp2_submit_headers,request when self dependency. https://github.com/nghttp2/nghttp2/commit/8716dd05d44f3b4cf0ff719240297cec57359815.
[34] Set max number of outgoing concurrent streams to 100 by default. https://github.com/nghttp2/nghttp2/commit/a151a44caf92d8bc7ecca8d8ec4780fa6206be96.
[35] Spark MLlib. https://spark.apache.org/mllib/.
[36] The DROWN Attack. https://drownattack.com.
[37] The Heartbleed Bug. http://heartbleed.com.
[38] Use NGHTTP2_PROTOCOL_ERROR when peer exceeds MAX_CONCURRENT_STREAMS limit. https://github.com/nghttp2/nghttp2/commit/e2bbc9461618d953e60c51f6ad3c44a65c178db5.
[39] Weak Diffie-Hellman and the Logjam Attack. https://weakdh.org.
[40] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*, volume 2. Addison-wesley Reading, 2007.
[41] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.
[42] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium*, NDSS '14, 2014.
[43] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC '14, 2014.
[44] Y. Bengio. Deep Learning of Representations for Unsupervised and Transfer Learning. In *Proceedings of the 2011 International Conference on Unsupervised and Transfer Learning Workshop - Volume 27*, UTLW '11, 2011.
[45] K. Bhargavan, D. Obradovic, and C. A. Gunter. Formal Verification of Standards for Distance Vector Routing Protocols. *Journal of the ACM*, 2002.
[46] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Rigorous Specification and Conformance Testing Techniques for Network Protocols, as Applied to TCP, UDP, and Sockets. *SIGCOMM*, 2005.
[47] J. A. Boyan and M. L. Littman. Packet Routing in Dynamically Changing Networks: A Reinforcement Learning Approach. In *Proceedings of the 6th International Conference on Neural Information Processing Systems*, NIPS '93, 1993.
[48] M. Burke and R. Cytron. Interprocedural Dependence Analysis and Parallelization. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '86, 1986.
[49] C. Cadar, D. Dunbar, and D. Engler. 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*, OSDI '08, 2008.
[50] R. Chang, G. Jiang, F. Ivancic, S. Sankaranarayanan, and V. Shmatikov. Inputs of Coma: Static Detection of Denial-of-Service Vulnerabilities. In *CSF*, 2009.
[51] Q. A. Chen, Z. Qian, Y. Jia, Y. Shao, and Z. M. Mao. Static Detection of Packet Injection Vulnerabilities – A Case for Identifying Attacker-controlled Implicit Information Leaks. In *ACM CCS*, 2015.
[52] B. Chess and G. McGraw. Static Analysis for Security. In *IEEE Security & Privacy*, 2004.
[53] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang. Neural Nets Can Learn Function Type Signatures From Binaries. In *Proceedings of the 26th USENIX Conference on Security Symposium*, Security '17, 2017.
[54] C. Cifuentes and M. V. Emmerik. Recovery of Jump Table Case Statements from Binary Code. In *Proceedings of the 7th International Workshop on Program Comprehension*, IWPC '99, 1999.
[55] J. Clause, W. Li, and A. Orso. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, 2007.
[56] B. De Sutter, B. De Bus, K. De Bosschere, P. Keyngnaert, and B. Demoen. On the Static Analysis of Indirect Control Transfers in Binaries. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 2000.
[57] Z. Deng, X. Zhang, and D. Xu. BISTRO: Binary Component Extraction and Embedding for Software Security Applications. In *Proceedings of the 18th European Symposium on Research in Computer Security*, ESORICS '13, 2013.
[58] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST Standard for Role-based Access Control. *ACM Transactions on Information and System Security (TISSEC)*, 2001.
[59] A. Flexeder, B. Mihaila, M. Petter, and H. Seidl. Interprocedural Control Flow Reconstruction. In *Proceedings of the 8th Asian Symposium on Programming Languages and Systems*, APLAS 2010, 2010.
[60] Y. Goldberg. A Primer on Neural Network Models for Natural Language Processing. *CoRR*, 2015.
[61] B. Hardekopf and C. Lin. Flow-Sensitive Pointer Analysis for Millions of Lines of Code. In *Proceedings of the 9th Annual International Symposium on Code Generation and Optimization*, CGO '11, 2011.
[62] HexHive. libdetox: Fast and efficient binary translator. https://github.com/HexHive/libdetox.
[63] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, et al. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. In *Instruction-Level Parallelism*, 1993.
[64] A. Javaid, Q. Niyaz, W. Sun, and M. Alam. A Deep Learning Approach for Network Intrusion Detection System. In *Proceedings of the 9th International Conference on Bio-inspired Information and Communications Technologies*, BICT '15, 2015.
[65] Y. Jiang, D. Wu, and P. Liu. JRed: Program Customization and Bloatware Mitigation Based on Static Analysis. In *2016 IEEE 40th Annual Computer Software and Applications Conference*, COMPSAC '16, 2016.
[66] Y. Jiang, C. Zhang, D. Wu, and P. Liu. Feature-based Software Customization: Preliminary Analysis, Formalization, and Methods. In *Proceedings of the 17th IEEE High Assurance Systems Engineering Symposium*, HASE '16, 2016.
[67] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu. Automated Concurrency-Bug Fixing. In *OSDI*, 2012.
[68] J. Kinder, F. Zuleger, and H. Veith. An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, 2009.

[69] N. Kothari, R. Mahajan, T. Millstein, R. Govindan, and M. Musuvathi. Finding Protocol Manipulation Attacks. In *SIGCOMM*, 2011.

[70] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, 2004.

[71] W. Lee and D. Xiang. Information-Theoretic Measures for Anomaly Detection. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, SP '01, 2001.

[72] K. R. M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, 2010.

[73] V. B. Livshits and M. S. Lam. Tracking Pointers with Path and Context Sensitivity for Bug Detection in C Programs. In *Proceedings of the 9th European Software Engineering Conference*, ESEC '11, 2003.

[74] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, 2005.

[75] C. D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[76] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. Resource Management with Deep Reinforcement Learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, 2016.

[77] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C prhttps://www.readcube.com/homeograms. In *CC*, 2002.

[78] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. WHYPER: Towards Automating Risk Assessment of Mobile Applications. In *USENIX security*, 2013.

[79] L. Pedrosa, A. Fogel, N. Kothari, R. Govindan, R. Mahajan, and T. Millstein. Analyzing protocol implementations for interoperability. In *NSDI*, 2015.

[80] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Using Probabilistic Generative Models for Ranking Risks of Android Apps. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, 2012.

[81] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen. Autocog: Measuring the Description-to-permission Fidelity in Android Applications. In *ACM CCS*, 2014.

[82] A. Quinn, D. Devecsery, P. M. Chen, and J. Flinn. JetStream: Cluster-scale Parallelization of Information Flow Queries. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, 2016.

[83] D. A. Ramos and D. R. Engler. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *USENIX Security Symposium*, 2015.

[84] T. Reps, S. Horwitz, and M. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, 1995.

[85] R. Saint-Germain. Information Security Management Best Practice Based on ISO/IEC 17799. *ARMA International Information Management*, 2005.

[86] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based Access Control Models. *IEEE Computer*, 1996.

[87] J. Saxe and K. Berlin. Deep Neural Network Based Malware Detection Using Two Dimensional Binary Program Features. In *Proceedings of the 10th International Conference on Malicious and Unwanted Software*, MALWARE '15, 2015.

[88] E. C. R. Shin, D. Song, and R. Moazzezi. Recognizing Functions in Binaries with Neural Networks. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC '15, 2015.

[89] 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 *IEEE Symposium on Security and Privacy*, 2016.

[90] 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 *Proceedings of the 2016 IEEE Symposium on Security and Privacy*, SP '16, 2016.

[91] R. Socher, Y. Bengio, and C. D. Manning. Deep Learning for NLP (Without Magic). In *Tutorial Abstracts of ACL 2012*, ACL '12, 2012.

[92] R. Sommer and V. Paxson. Outside the Closed World: On Using Machine Learning for Network Intrusion Detection. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, 2010.

[93] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Information systems security*, 2008.

[94] Y. Sui and J. Xue. SVF: Interprocedural Static Value-flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, 2016.

[95] H. Theiling. Extracting safe and precise controlfl ow from binaries. In *Proceedings of the 7th International Workshop on Real-Time Computing and Applications Symposium*, RTCSA 2000, 2000.

[96] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *NDSS*, 2000.

[97] R. Wang, W. Enck, D. Reeves, X. Zhang, P. Ning, D. Xu, W. Zhou, and A. M. Azab. EASEAndroid: Automatic Policy Analysis and Refinement for Security Enhanced Android via Large-scale Semi-Supervised Learning. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, 2015.

[98] S. Wang, P. Wang, and D. Wu. Reassembleable Disassembling. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC '15, 2015.

[99] S. Wang, P. Wang, and D. Wu. UROBOROS: Instrumenting Stripped Binaries with Static Reassembling. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering*, SANER '16, 2016.

[100] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy. Early Detection of Configuration Errors to Reduce Failure Damage. In *OSDI*, 2016.

[101] E. Yuan and J. Tong. Attributed based Access Control (ABAC) for Web Services. In *IEEE ICWS*, 2005.

[102] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. ConSeq: Detecting Concurrency Bugs Through Sequential Errors. In *ACM SIGPLAN Notices*, 2011.

[103] Z. Zhu and T. Dumitras. FeatureSmith: Automatically Engineering Features for Malware Detection by Mining the Security Literature. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, 2016.