Automatic Feature Isolation in Network Protocol Software Implementations

Ze Zhang
University of Michigan
Ann Arbor, Michigan, USA
zezhang@umich.edu

Qingzhao Zhang
University of Michigan
Ann Arbor, Michigan, USA
qzzhang@umich.edu

Brandon Nguyen
University of Michigan
Ann Arbor, Michigan, USA
brng@umich.edu

Sanjay Sri Vallabh Singapuram
University of Michigan
Ann Arbor, Michigan, USA
singam@umich.edu

Z. Morley Mao
University of Michigan
Ann Arbor, Michigan, USA
zmao@umich.edu

Scott Mahlke
University of Michigan
Ann Arbor, Michigan, USA
mahlke@umich.edu

ABSTRACT

Common vulnerabilities and exposures (CVE) usually exploit design or implementation flaws of specific features in widely used network protocols, causing serious security and safety threats on full-scale devices and systems. Feature isolation as a general protocol customization practice is shown to be highly promising to reduce attack surfaces in these protocols. In this work-in-progress paper, we propose a systematic approach to achieve automatic feature isolation using various program analysis based techniques. Specifically, we present two methods targeting different feature granularity to automatically identify and isolate unnecessary features in a software protocol implementation. In addition, we develop a semantic reconstruction mechanism to enforce user-specified feature access control policies. Preliminary case studies confirm that our proposed techniques can be effectively applied on real-world protocol vulnerabilities.

CCS CONCEPTS

• **Software and its engineering** → **Automated static analysis**; • **Networks** → *Protocol testing and verification.*

KEYWORDS

static program analysis; network protocol customization; security;

ACM Reference Format:
Ze Zhang, Qingzhao Zhang, Brandon Nguyen, Sanjay Sri Vallabh Singapuram, Z. Morley Mao, and Scott Mahlke. 2020. Automatic Feature Isolation in Network Protocol Software Implementations. In *2020 Workshop on Forming an Ecosystem Around Software Transformation (FEAST'20), November 13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3411502.3418425

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FEAST'20, November 13, 2020, Virtual Event, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8089-8/20/11…$15.00
https://doi.org/10.1145/3411502.3418425

1 INTRODUCTION

With the increasing demands of network services, numerous large-scale systems as well as billions of mobile and IoT devices are running network protocols [3, 36] to exchange information and requests. However, severe malicious attacks have also been engineered to threaten these commonly used protocols, resulting in massive information stealing and notorious ransomware [1, 35, 39]. Fixing exposed vulnerabilities requires significant amount of effort from developers because they need to comprehensively understand the latest protocol details before making any change. In addition, protocol upgrade is also a challenging task for production servers as many of them must keep high availability and cannot afford the frequent upgrade interruption. Although both research and industry communities have developed various techniques to harden network and application protocols [6, 44], effectively preventing protocol vulnerabilities from external threats still remains an open question.

According to the previous study [30], we find that a majority of malicious attacks exploit design or implementation flaws of a specific feature or extension presented in a network protocol. In order to make the protocol broadly applicable, many of them are implemented as a one-size-fits-all library or software package, such as SSL/TLS, HTTP, and SSH. Although these embedded features benefit specific tasks, many of them become redundant and may never be invoked in real use scenarios. More dangerously, these extraneous features become potential attack surfaces that could have been prevented. One classic example is the HeartBleed [39] security bug resulting from an implementation flaw in the OpenSSL cryptography library [36]. Many applications do not use the vulnerable heartbeat feature but it is turned on by default in the older version of OpenSSL. To avoid this risky situation there is a need to provide a practical solution for the identification and isolation of unnecessary features.

Feature isolation is a set of techniques to identify and disable redundant features included in standard protocol implementations. Its goal is to automatically generate a customized protocol implementation that only contains required functionalities without changing its original run-time behaviors. In this way, we can effectively reduce the attack surface and eliminate vulnerabilities associated with unused features. Unfortunately, the current feature isolation process is mostly performed by developers in an ad-hoc manner and

is far from systematic. Their solutions usually rely on predefined macros or compilation flags which are difficult for achieving the optimal solution between security and usability.

To address this limitation, we aim to provide a systematic and sufficiently automated approach to reduce the attack surface of network protocols through a set of program analysis based feature isolation techniques. Our proposed system collects necessary inputs from users as well as protocol specifications, and generates a customized protocol implementation with unnecessary features removed. In addition, our system also performs required validations to make sure the customization does not change protocols' run-time behaviors. Although the entire system is still a work in progress, this paper presents the core techniques that we used to achieve feature isolation. First, we develop a block-level isolation technique that can redirect the protocol execution path to separate feature related code regions. Second, in order to provide fine-grained modifications, we also design an instruction-level isolation method to skip specific instructions or function calls related to unwanted features. Lastly, we investigate a semantic reconstruction method to enforce proper conditions for feature access control. Using real-world protocol vulnerabilities, our preliminary case studies prove the effectiveness of our proposed techniques.

2 BACKGROUND AND MOTIVATION

2.1 Common Vulnerabilities and Exposures

We study 18 medium to high impact Common Vulnerabilities and Exposures (CVEs) found in commonly used network protocols in the past decade and conclude that all of these vulnerabilities are caused by some forms of design or implementation flaws in protocol features or extensions, most of which are rarely used but are enabled by default. Table 1 summarizes different feature isolation approaches for each CVE we examined, which can further be addressed by our proposed techniques. Besides, program analysis has been proven a powerful tool for systematically understanding and solving various computer software problems, such as code optimization [2, 31], software bug detection [41, 42], and reliability [37, 43]. More importantly, program analysis techniques have also been extensively applied in analyzing protocol security and correctness [5, 6, 33, 38]. These previous successes in security areas suggest that program analysis can be highly useful for addressing protocol feature isolation challenges.

2.2 Feature Isolation

Feature isolation (also known as feature debloating) techniques have been studied in previous works. CARVE [4] deploys annotations to map software features to source code, but the annotation is at the source code level, making it difficult to use if one is not familiar with the code base. CHISEL [29] compiles the source code and executes it with different test cases to perform a reinforcement learning-based approach. TOSS [7] relies on dynamic tainting techniques to locate feature-related code and applies binary rewriting to remove redundant features. However, both of them may introduce new vulnerabilities because of the dynamic feature identification process. In this work, we explore static program analysis solutions for feature isolation. Similar to our idea, Jred [32] is also a static analysis tool to trim unused code for reducing attack surfaces, but it operates on Java applications and run-time environments. Our work targets on commonly used network protocols which are implemented in C/C++ instead.

Feature Isolation Approach	CVEs
Disabling via global variable	Heartbleed [14], HTTP_PROXY redirection [24]
Disabling via run-time variable	HTTP_PROXY redirection [24], Apache integer overflow [10], XMPP dialback [21], XMPP message carbons [26], OpenSSH information leak [20], HTTP proxy data leak [28], Apache XSS [11], FREAK [16], Logjam [18], RC4 [17], CRIME [12], BREACH [13]
Adding customized access control	Slow read DoS [23], Apache integer overflow [10], Range header DoS [9], Dependency cycle DoS [19], HPACK bomb [22, 25], HTTP proxy DoS [8]

Table 1: Feature isolation approaches from CVE study.

3 SYSTEM DESIGN

3.1 Problem Statement

Feature isolation aims to customize a standard protocol implementation to a specialized one which can completely or conditionally disable extraneous features. From the implementation perspective, a feature is usually a group of instructions distributed in different functions that performs a specific functionality. Overall, there are two fundamental requirements in this process. First, feature isolation should be complete. If one feature is disabled, there is no possible execution path triggering that feature in the customized software. Second, feature isolation must be safe. Except for unwanted features, all other functionalities should be unaffected and the protocol can still executes as normal. We will introduce a motivating example, *disabling Apache HTTP Server .htaccess*, to illustrate the feature isolation problem.

.htaccess is a type of configuration setting in Apache HTTP Server [3]. On each incoming request, *httpd* prepares a data structure containing the request information, tries to find *.htaccess* files, processes the content of these files, and finally updates the configuration. Unfortunately, it also allows attackers to maliciously change configurations, as described in CVE-2017-9798 [27]. Disabling the usage of *.htaccess* for non-privileged users can prevent this security vulnerability. As shown in Figure 1, the *.htaccess* feature is implemented across 8 different functions involving around 850 lines of code (LOC). It involves hook mechanism, writes on core parameters and massive data flows. Isolating the entire *.htaccess* feature in a complete and safe manner is challenging because of following reasons:

First, since the *.htaccess* feature is distributed in many different correlative program blocks, we need to systematically locate and then modify various program points to guarantee the feature is completely isolated. Moreover, it is crucial to understand the relation among these functions so that we can correctly handle program interactions such as the hook registration, *.htaccess* file opening, and the data processing simultaneously.

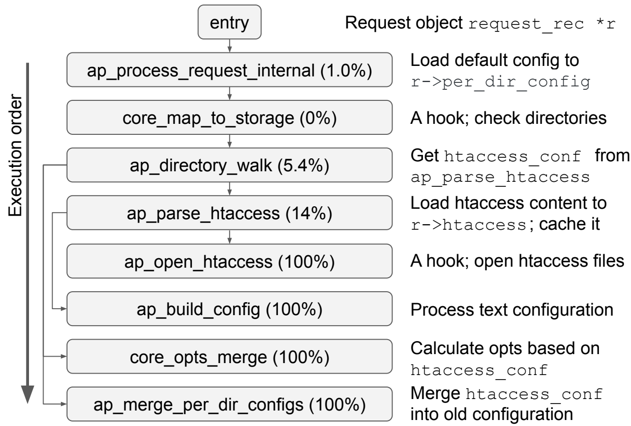

Figure 1: Function call graph of Apache HTTP Server .htaccess feature (*httpd 2.4.43*). The functions are ordered by execution sequence and brief comments about the functions' data flow are illustrated on the right side. The percentage shown next to each function name represents the coverage of feature related code at the basic block level (i.e., blocks executed by test cases triggering the .htaccess feature but not visited by test cases without the feature).

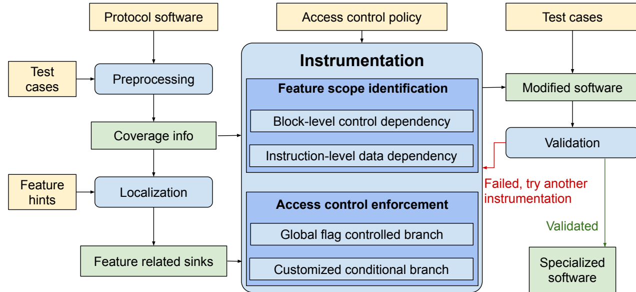

Figure 2: Overview of feature isolation system. Yellow boxes represent user inputs, blue boxes are instrumented modules and green boxes are outputs.

Second, we find that naively deleting feature related code violates data dependencies and results in critical program errors because the *.htaccess* feature is closely intertwined with other features. Considering that users usually do not fully understand the protocol implementation nor provide detailed annotations, automatic feature isolation requires detailed program analysis to intelligently decouple the targeted feature and fix corresponding dependency issues.

Third, the isolation approach heavily depends on the feature implementation logic given control flows and data flows in the program. For example, it is possible to either disable the whole code block handling *.htaccess* files or only remove the specific function call to *ap_parse_htaccess*. To improve the effectiveness and the applicability, we need to design and apply different isolation techniques based on feature code-level representations.

3.2 System Overview

In this section, we present a high-level overview of our proposed automatic feature isolation system. As shown in Figure 2, the system takes four inputs: protocol software as the subject, feature hint for identifying unwanted features, access control policy representing the expected result of feature isolation, and test cases for program profiling as well as validation. The final output is a specialized protocol software implementation where the unnecessary features are controlled by access control policies. The entire system is consisted of following modules:

The *preproccessing* module takes the protocol software and executes test cases to profiles their execution coverage (i.e., visited basic blocks and program call graphs etc.). The test cases are divided into two categories: positive test cases execute normal functionalities which should be preserved after the feature customization and negative test cases try to trigger removed features to examine the outcome.

The *localization* module identifies related "sinks", defined as call instructions, global stores, as well as variable assignments that modify the program state to implement a feature among the whole protocol software. The localization process takes both feature hints and coverage information as its inputs. The feature hint can be a string pattern or other annotations used as a clue for finding feature related code. For the htaccess case mentioned in Section 3.1, an input string "htaccess" can help identify two feature-related functions shown in Figure 1. The coverage information further indicates that the code regions that are covered by negative test cases but are not included in positive test cases are likely to be feature related. Combining two results, localization module ranks in-code instructions and outputs the sinks that are most likely to be associated with the target feature.

The *instrumentation* module applies static program analysis on the protocol software and modifies it through code instrumentation, which has two main steps. First, the module needs to pinpoint which part of the code is related to targeted sinks. To do this, the system constructs the call graph as depicted in Figure 1, performs data flow analysis, and labels all instructions or functions that have dependency with the sink. Then the system applies either block-level or instruction-level analysis (or both) to decide which code blocks or specific instructions should be wrapped by access control policies. The choice between the block-level and the instruction-level isolation depends on implementation details of the original software and each of them has its own advantages as discussed in Section 3.3 and Section 3.4. Second, access control enforcement is performed to transform user-defined access control policies into specific code instrumentation. For example, it may need to insert some checking conditions or run-time constraints at the entry point of feature-related regions to redirect the program control flow. The coverage information generated from preprocessing module also guides the instrumentation.

Finally, the *validation* module uses the test cases to verify whether the feature isolation in the modified software is complete (i.e. all negative test cases fail) and safe (i.e. all positive test cases pass). If the modified implementation does not meet all verification requirements, the system will go back to the instrumentation module, mutate the instrumentation plan, and apply the new instrumentation on the original software. This process repeats until one modified version passes all validations.

Our proposed system works on the LLVM compiler infrastructure [34]. The protocol software is firstly converted into LLVM

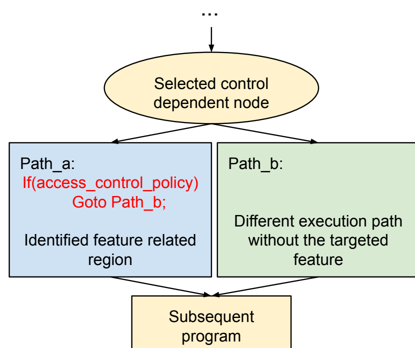

Figure 3: High-level operation of block-level isolation technique.

Intermediate Representation (IR), which allows us to apply a series of static analysis and code instrumentation passes. Note that the *preprocessing* module uses the information generated from the Gcov-based profiling [40]. In our preliminary implementation, the feature hints consist of patterns of IR instructions as well as predefined strings to represent a feature at high level. In addition, test cases used for profiling and validation are either from public repositories or written manually.

3.3 Block-level Isolation

For features either associated with large code regions (present in multiple sequential basic blocks) or already implemented with disabling mechanisms, we propose to use block-level isolation method to redirect the protocol execution flow to a different path without the related feature. This method is built on top of the control dependency analysis which allows us to find a set of control dependent nodes triggering the targeted feature. With this information, we can correspondingly find the execution path without involving the feature we want to remove. Since block-level isolation skips one or more basic blocks in the protocol implementation, a substantial number of instructions will not be executed. Therefore, we need to make sure that all instructions included in these basic blocks are part of the feature and it is safe to skip them.

Our current implementation involves an inter-procedural control dependency analysis. Starting from the specified entry point (e.g. a function annotated by the sink), an inter-procedural control flow graph (ICFG) is first generated by recursively plotting the CFG of each callee function invoked in the entry function. With the ICFG, we can then perform the control dependency analysis to collect all the conditional branches (control dependent nodes) as well as their corresponding branch values that lead to the feature to be isolated. By picking a control dependent node, we are able to alter the protocol execution flow to a different branch direction without involving the targeted feature. The entire procedure is summarized in Figure 3, where the red texts illustrate the instrumented instructions at the beginning of the identified feature related region. The details of access control policy will be explained in section 3.5.

To make sure the entire feature-related region is safe to skip, we need to check that there is no instruction related to other features: based on the profiling information, we first find the execution difference between positive and negative test cases. This execution difference is then used as a guide to determine which instructions are related to the target feature. In addition, we trace the data flow of each variable assignment in this region. If all the variable uses are within the same region, implying the temporary local variable, we can then infer that separating the identified region does not have a negative impact on other parts of the protocol. Currently, we use the coverage information produced by the preprocessing module to assist this process. We are also investigating other methods such as code-level tainting [4] to further improve the accuracy. Note that if multiple control dependent nodes can be used to isolate the feature, we will select the closest node to ensure the minimum modification on the original software.

If feature-related blocks are on the critical path, we will not find any existing control dependent node to redirect the protocol execution path. In this case, we need to build and insert our own control dependent node, but how to determine the start and end points to achieve the optimal solution is still an ongoing work. In case it is not safe to skip the entire region, we also provide a fine-grained isolation mechanism, as described in the next section.

3.4 Instruction-level Isolation

For features that are associated with the individual store instructions of variable assignments or function calls, we propose the use of instruction-level isolation. This method allows for the selective skipping of these feature-related instructions. For example, a feature might be disabled by skipping a call to a particular function. Such a function may have a return value that is checked and instruction-level customization serves as the mechanism to conditionally skip the function call while ensuring that following uses of the return value have a valid value.

While this may seem trivial on the surface, the repairing of data flow to maintain correct execution presents an interesting problem. Variable assignments and function calls may provide values for uninitialized variables by writing to pointers/references passed in as arguments or storing to variables: skipping these will result in future uses of the variable being with uninitialized values and result in incorrect program behavior. Determining what value should be stored in absence of the original instruction is a challenge problem since integers and pointers can take on billions of different values. Normally a person familiar with the code base would have to provide such information, but the automatic discovery and generation of default values is an interesting research question.

We define the term state-modifying instruction to refer to stores and function calls that has return values and/or writes to pointer arguments. At a high level, our method takes feature-related state-modifying instructions and replaces them with a conditional branch leading to two blocks. The conditional branch checks to see if the feature is enabled with one block serving as the “feature-enabled” path with the state-modifying instructions and the other block serving as the “feature-disabled, fix-up” path that performs data flow repairs in absence of the state-modifying instructions. These two blocks converge to the instructions that follow the feature-related instructions. The high-level transformation of this method is illustrated in Figure 4.

The fix-up path is where the biggest problem lies: what values should the variables take on in absence of assignments? Due to the use of null pointers to indicate lack of a pointed-to resource,

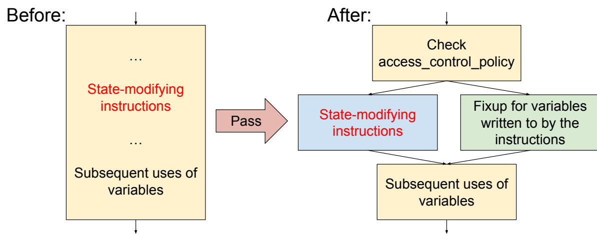

Figure 4: High-level operation of instruction-level isolation technique.

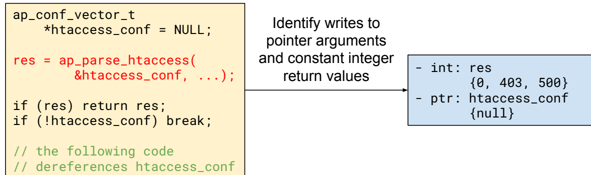

Figure 5: Example with *.htaccess*. The instruction-level isolation identifies that `ap_parse_htaccess` writes to `htaccess_conf` and thus will be handled in fixup path. The technique finds that `res` can only take on three values. These values can be used for further analysis such as examining their use as a branch condition in the following `if` statement.

we decide to assign pointers to null as the first step. Currently we focus on a subset of integer values: integer literals returned by a function and not values that are computed such as those resulting from arithmetic operations. By focusing on integer literals we can enumerate all possible return values from a function, reducing the number of different values that an integer can take on from billions to only the ones that the function is able to return and thus provide a starting point for analysis. This is illustrated in Figure 5 using the call from `ap_directory_walk` to `ap_parse_htaccess` for the *.htaccess* feature. Currently we are investigating approaches to utilize this set of return values to inform which value should be assigned to the variable in the fix-up path. One such approach involves the analysis of uses of the variables after the state-modifying in branch conditions. The more general problem of determining the proper value of normal integers and pointers is still an open research problem.

3.5 Access Control Enforcement

The access control policy is represented as a Boolean expression and is used to conditionally trigger an isolated feature, depending on user specified inputs such as external settings (e.g. command line options) or run-time constraints (e.g. length of a string). Our goal is to automatically generate these policies and insert them in appropriate places. However, the code generation for the access control policy happens in LLVM IR where the source-code level information is absent, which brings two challenges:

The first one is with the availability of variables appearing in the access control policy. If a policy cannot be constructed with all variables available within the scope, a global variable needs to be introduced across all LLVM modules. Unfortunately, declaring global variables may lead to unintentional side-effects, including

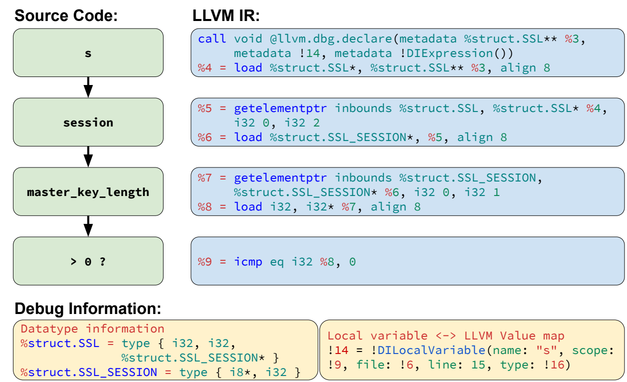

Figure 6: Semantic reconstruction with LLVM IR and debug information.

namespace pollution or concurrency issues, especially when modules are linked dynamically. Therefore, introducing variables into the global scope must be done with extra care.

The other challenge is with mapping high-level Boolean expressions with corresponding instructions in low-level LLVM IR. This is due to the absence of association between code-level variables and temporary LLVM registers used in the static single assignment (SSA) form. In addition, compiler optimizations such as common sub-expression elimination and loop-invariant code motion can further reorder or modify instructions, making it difficult to reconstruct the original expression from the modified IR.

Our current implementation aims to semantically reconstruct the high-level access control expressions with appropriate LLVM IR instructions to generate the branch condition for isolated features. To identify variables used in the policy, we utilize the source-level debug information preserved in LLVM IR files by using a debug-build of the code base. The debug information maintains the association of high-level variables with their corresponding LLVM temporary registers. We also use the datatype information available in this metadata to calculate appropriate offsets when accessing pointer references and structure fields. Given an expression involving nested dereferencing of structure fields (e.g. `a->b->c`), our mechanism to code-generate accesses to structure fields is applied repeatedly until the access to the deepest field is generated. Since the access control policy usually consists of multiple expressions to compose a complex triggering condition, the same mechanism is recursively applied to all operands presented in the policy, which in turn could be nested expressions. Once the mappings of high-level variables to LLVM temporary registers are available, they are fed into appropriate comparison instructions reflecting the high-level Boolean operation in the policy. The final result of code-generated instructions is consumed by the conditional branch instruction that gates the control-flow into the isolated code block. Overall, this process semantically reconstructs the high-level Boolean expression using low-level LLVM IR instructions. The modified IR can then be rebuilt, after dropping the debug information to save space and being optimized for run-time performance.

Consider the following example of enforcing the access control policy `"s->session-> master_key_length > 0"`, which can be used to fix the *ChangeCipherSpec* CVE [15]. As shown in Figure

6, the first step would be to look up the LLVM temporary register associated with the variable "s" from the debug information. To code-generate the access to the sub-field "session", the offset of this field is inferred from the composite datatype pointed to by "s". Then, the access to the child-field "master_key_length" is in turn generated by looking up the corresponding offset of the field from the pointer resulting from the previous step. Finally, an integer comparison instruction is inserted to check whether the value from the previous step is greater than 0.

However, the proposed code generation mechanism still faces a limitation since the access control policy needs to be expressed in terms of the source code. If users are not familiar with the software implementation, it is hard for them to define low-level policies based on the source code. To solve this, we are planning to define a domain specific language that allows users to construct their policies at a higher level abstracted away from the implementation details.

4 CASE STUDY

Our case studies focus on Apache HTTP Server [3] and OpenSSL [36] because both of them are widely used protocol implementations with lots of features and CVE reports. Our findings confirm the need for automatic feature isolation support to prevent protocol vulnerabilities and real-world threats. For example, in Apache HTTP Server, we can change the *mod_http2* parameter in the configuration file to either enable or disable all HTTP2 related features. However, this default approach does not provide any flexibility if users only need part of the HTTP2 features. Our proposed feature isolation techniques make it possible to selectively enable certain required features while disabling the rest for reducing attack surfaces, all the while keeping HTTP2 operating as normal. In the older version of OpenSSL (1.0.1f), we find at least 97 compiler flags defined by developers to disable specific protocol features. According to our analysis, these flags appear around 2460 times in 415 source files across the whole codebase, making the automatic feature isolation a useful but challenging practice. More specifically, we will present two concrete feature isolation tasks, removing the *.htaccess* in Apache HTTP Server and disabling *ChangeCipherSpec* with a zero-length master key in OpenSSL, to validate the effectiveness of our proposed techniques.

***.htaccess* in Apache HTTP Server.** As mentioned in Section 3.1, our goal is to prevent the Apache HTTP Server from loading the *.htaccess* file to eliminate its specific attack surface [27]. In this case, the feature hint provided from users is the string "htaccess". The system searches the codebase and finds out two functions, *ap_parse_htaccess* and *ap_open_htaccess*, that are likely to be related to *.htaccess* feature as their function names suggest. The *localization* module then finds and labels call instructions to above functions as sinks. To disable this feature, both block-level and instruction-level isolation mechanisms are used. Function *ap_directory_walk* has a do-while loop handling *.htaccess* files and all instructions in this loop are related to the *.htaccess* feature. Disabling this code region requires changing the program control flow and breaking the loop, which makes the block-level isolation a perfect candidate. Our inter-procedural control dependency analysis finds the conditional branch instruction entering the loop body and inserts new control flows to make sure the loop is never executed under the specified access control policy. However, in function *register_hooks*, the feature hook function call to *ap_hook_open_htaccess* lies in the same basic block with other function calls. Since we only want to skip a targeted instruction, we should perform instruction-level isolation so that other features are not affected. The instruction-level isolation will analyze calls to *ap_hook_open_htaccess* and find that it returns no value nor stores to any pointer argument. This results in a branch path illustrated in Figure 4 where the fix-up path consists only of a branch to converge back to the instruction that initially followed the call to *ap_hook_open_htaccess.*

Since we want to permanently remove this feature for certain user groups, the access control policy simply allocates a global flag and asks administrators to switch on/off the *.htaccess* feature by controlling the flag. To verify our instrumentation, we write a negative test case that tries to invoke the *.htaccess* feature and the test case fails on the modified software, implying the feature is effectively disabled.

ChangeCipherSpec in OpenSSL with a zero-length master key. This case is constructed from a real world vulnerability CVE-2014-0224 [15], which is a man-in-the-middle attack triggered by abnormal *ChangeCipherSpec* messages with a zero-length master key. Our goal is to disable the processing of *ChangeCipherSpec* messages when there is a potential attack (e.g. the length of the master key is zero). The protocol software we used in this example is OpenSSL 1.0.1g, the version before the vulnerability is fixed. The feature hint is assignments to the variable *SSL.session.cipher* which represents the current cipher spec. Through the *localization* module, the sink is found in function *ssl3_do_change_cipher_spec.* We choose to perform the block-level isolation because the sink is the only instruction in its basic block. However, this case differs from the *.htaccess* because of its access control policy, which inserts a run-time checking on the length of master key (variable *SSL.session.master_key_length*) to make sure it is greater than zero before processing the *ChangeCipherSpec* messages, as illustrated in Figure 6. Our automatic policy generation is consistent with the fix performed by OpenSSL developers (Github commit *a7c682fb*) and is validated by negative test cases.

5 CONCLUSION

Many network protocols are packaged as one-size-fits-all libraries and include rarely used features or extensions which could become potential attack surfaces, resulting in security concerns. In this work-in-progress paper, we present two static program analysis based methods targeting different feature granularity to automatically identify and isolate redundant features in network protocol implementations. In addition, we also develop a policy enforcement mechanism to instrument user specified run-time constraints. Preliminary case studies on protocol vulnerabilities prove the effectiveness of our proposed solutions.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their constructive comments for improving this work. This research is supported by the Office of Naval Research under the grant N00014-18-1-2020.

REFERENCES

[1] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, et al. 2015. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 5–17.

[2] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph.D. Dissertation. University of Cophenhagen.

[3] APACHE. 2020. Apache HTTP Server Project. https://httpd.apache.org/.

[4] Michael D Brown and Santosh Pande. 2019. CARVE: Practical security-focused software debloating using simple feature set mappings. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*. 1–7.

[5] Richard Chang, Guofei Jiang, Franjo Ivancic, Sriram Sankaranarayanan, and Vitaly Shmatikov. 2009. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *2009 22nd IEEE Computer Security Foundations Symposium*. IEEE, 186–199.

[6] Qi Alfred Chen, Zhiyun Qian, Yunhan Jack Jia, Yuru Shao, and Zhuoqing Morley Mao. 2015. Static detection of packet injection vulnerabilities: A case for identifying attacker-controlled implicit information leaks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 388–400.

[7] 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*. 1–7.

[8] The MITRE Corporation. 2008. CVE-2008-2364: mod_proxy_http DoS. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-2364.

[9] The MITRE Corporation. 2011. CVE-2011-3192: Range header DoS. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-3192.

[10] The MITRE Corporation. 2011. CVE-2011-3607: Integer overflow in Apache HTTP server. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-3607.

[11] The MITRE Corporation. 2012. CVE-2012-3499: Apache XSS. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-3499.

[12] The MITRE Corporation. 2012. CVE-2012-4929: CRIME attack. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-4929.

[13] The MITRE Corporation. 2013. CVE-2013-3587: BREACH attack. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-3587.

[14] The MITRE Corporation. 2014. CVE-2014-0160: Heartbleed bug. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160.

[15] The MITRE Corporation. 2014. CVE-2014-0224. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0224.

[16] The MITRE Corporation. 2015. CVE-2015-0204: OpenSSL FREAK attack. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0204.

[17] The MITRE Corporation. 2015. CVE-2015-2808: RC4 attack. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-2808.

[18] The MITRE Corporation. 2015. CVE-2015-4000: Logjam attack. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-4000.

[19] The MITRE Corporation. 2015. CVE-2015-8659: Dependency cycle DoS. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8659.

[20] The MITRE Corporation. 2016. CVE-2016-0777: OpenSSH client information leak. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0777.

[21] The MITRE Corporation. 2016. CVE-2016-1232: Prosody XMPP dialback vulnerability. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-1232.

[22] The MITRE Corporation. 2016. CVE-2016-1544: HPACK bomb. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-1544.

[23] The MITRE Corporation. 2016. CVE-2016-1546: Slow read DoS. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-1546.

[24] The MITRE Corporation. 2016. CVE-2016-5387: HTTP_PROXY redirection. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5387.

[25] The MITRE Corporation. 2016. CVE-2016-6581: HPACK bomb. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-6581.

[26] The MITRE Corporation. 2017. CVE-2017-5858: XMPP Message Carbons extension vulnerability. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5858.

[27] The MITRE Corporation. 2017. CVE-2017-9798. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9798.

[28] The MITRE Corporation. 2019. CVE-2009-1191: mod_proxy_ajp data leak. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1191.

[29] 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*. 380–394.

[30] David Ke Hong, Qi Alfred Chen, and Z Morley Mao. 2017. An initial investigation of protocol customization. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*. 57–64.

[31] Wen-Mei W Hwu, Scott A Mahlke, William Y Chen, Pohua P Chang, Nancy J Warter, Roger A Bringmann, Roland G Ouellette, Richard E Hank, Tokuzo Kiyohara, Grant E Haab, et al. 1993. The superblock: An effective technique for VLIW and superscalar compilation. In *Instruction-Level Parallelism*. Springer, 229–248.

[32] Yufei Jiang, Dinghao Wu, and Peng Liu. 2016. Jred: Program customization and bloatware mitigation based on static analysis. In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 12–21.

[33] Nupur Kothari, Ratul Mahajan, Todd Millstein, Ramesh Govindan, and Madanlal Musuvathi. 2011. Finding protocol manipulation attacks. In *Proceedings of the ACM SIGCOMM 2011 conference*. 26–37.

[34] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE, 75–86.

[35] Sarah Madden and Christian Dresen. 2016. The DROWN Attack. https://drownattack.com.

[36] OpenSSL. 2020. OpenSSL Cryptography and SSL/TLS Toolkit. https://www.openssl.org/.

[37] Sunghyun Park, Shikai Li, Ze Zhang, and Scott Mahlke. 2020. Low-cost prediction-based fault protection strategy. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 30–42.

[38] Luis Pedrosa, Ari Fogel, Nupur Kothari, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. Analyzing protocol implementations for interoperability. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*. 485–498.

[39] Inc. Synopsys. 2020. The Heartbleed Bug. http://heartbleed.com.

[40] The Clang Team. 2020. Clang Compiler User's Manual: GCOV-based Profiling. https://clang.llvm.org/docs/UsersManual.html#gcov-based-profiling.

[41] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. 2016. Early detection of configuration errors to reduce failure damage. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 619–634.

[42] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. 2011. ConSeq: detecting concurrency bugs through sequential errors. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 251–264.

[43] Ze Zhang, Sunghyun Park, and Scott Mahlke. 2020. Path Sensitive Signatures for Control Flow Error Detection. In *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. 62–73.

[44] Ziyun Zhu and Tudor Dumitraş. 2016. 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*. 767–778.