TOSS: Tailoring Online Server Systems through Binary Feature Customization

Yurong Chen, Shaowen Sun, Tian Lan, Guru Venkataramani
{gabrielchen,sunshaowen,tlan,guruv}@gwu.edu

ABSTRACT

Network-based models are increasingly adopted to deliver key software service and utilities (e.g., data storage, search, and processing) to end users. The need to satisfy diverse user requirements and to fit different application environment often leads to continual expansion and addition of new (and in many cases excessive) features, known as the feature creep problem. Existing work mitigating feature bloat often either debloats programs at source code level (which may not always be available, in particular for legacy systems) or customize binary only with respect to very limited scope of inputs. In this paper, we propose a new approach, TOSS, for automated customization of online servers and software systems, which are implemented using a client-server architecture based on the underlying network protocols. Specifically, TOSS harnesses program tracing and tainting-guided symbolic execution to identify desired (feature-related) code from the original program binary, and apply static binary rewriting to remove redundant features and directly create customized program binary with only desired features. We implement a prototype of TOSS and evaluate its feasibility using real-world executables including Mosquito, which relies on the Message Queuing Telemetry Transport (MQTT) protocol for lightweight Internet of Things (IoT) communications. The results show that TOSS is able to create a functional program binary with only desired features and significantly reduce potential attack surface by eliminating undesired protocol/program features.

ACM Reference Format:

1 INTRODUCTION

Due to the need to satisfy diverse end user requirements and to suit different application environments, network-based protocols that are designed to deliver key service capabilities and utilities (e.g., data storage, search and processing) often lead to continual expansion and addition of new (and in many cases excessive) features, known as the feature creep problem [5]. Feature creep in online server systems not only results in larger installation footprint, but also causes an increased attack surface with higher possibility of vulnerabilities and exploitation. Real-world examples of protocol feature creep include the trap communication feature in the Simple Network Management Protocol (SNMP) and the heartbeat feature in the Open Secure Sockets Layer (OpenSSL), both of which may be unnecessary under most practical scenarios, but unfortunately cause serious security threats such as denial of service attack and leakage of sensitive information.

An effective approach to mitigate feature creep is debloating, e.g., creating customized software systems that contain just-enough features and yet satisfy specific user needs, in order to minimize the software complexity and resulting attack surface. Prior work on static software debloating [4, 5] is often conducted on source code (where redundant functions and features are relatively easier to identify) to remove unused code. However, source code may not always be available especially for commercial off-the-shelf (COTS) or legacy programs. With only program binaries available, even correctly recognizing function body itself is a challenging task [1]. It may be impossible to provide “seed functions” that are usually required by existing feature removal techniques to use as the source of slicing and to bootstrap the analysis process. On the other hand, new program binaries directly constructed (or extracted) from runtime traces (sometimes with additional static analysis) through the binary reuse technique [2, 6, 18, 23, 25] can only achieve correct execution with very limited scope of user inputs.

Main problem and challenges. In this paper, we propose a new approach, TOSS, for automated customization of online servers and software systems, which are often implemented using a client-server architecture based on the underlying network protocols, e.g., Message Queuing Telemetry Transport (MQTT) protocol for lightweight Internet of Things (IoT) communications. We define automated feature customization as the process of identifying and rewriting different software and service features from a binary executable. In many COTS and legacy software, source code may no longer be available. Hence customization of binary considered in this paper is more relevant. Without requiring any knowledge of potential exploits, the customized programs contain just-enough software features to support only the required services, thus significantly reducing the attack surface and the exposure to future exploitation through features (e.g., zero-day attacks). Our approach goes beyond existing work on feature separation [12], reduction [5] and code de-bloating [4, 19, 20], which focus on removing unused or unnecessary code. We argue that vigilantly managing and customizing permitted features is crucial for achieving improved software security [5], especially for online servers and software systems that are frequently targeted by attackers because of the value of their data and services.
At the core of automated customization, a key problem is to identify software features directly from a program’s binary, without access to source code or debug symbol information. We define a feature as a collection of basic blocks, which uniquely represent an independent, well-contained and stateless service or capability of the program. First, since features in an online server or a software system are often accessed via the network and triggered upon user requests, we employ dynamic tainting to monitor and track the execution of a target feature, utilizing the user request packets or relevant fields as taint sources. Second, since feature executions in dynamic analysis depend upon specific input values, a pure dynamic tainting approach may be difficult to achieve sufficient code coverage, and in other words, the identified program code can only handle very limited scope of user inputs related to the target features. To this end, we harness dynamic tracing and symbolic execution - while dynamic tracing can reveal the basic blocks that are required in specific feature executions, we utilize dynamic tainting to guide an symbolic execution engine to effectively discover more program code and execution paths related to the target features. As a result, TOSS delivers a customized program that has the unique ability to correctly execute desired features with a wide range of feasible user inputs.

Identifying feature-related code segments enables us to rewrite program features, in accordance with user needs. We leverage binary rewriting tools such as DynInst to obtain a customized binary that only retains the desired program features while removing the other redundant ones. In particular, we replace undesired basic blocks with “NOP”s to eliminate unwanted features, and in case such features/basic blocks are still invoked during program execution, we further redirect their invoking instruction such as function call or jump to a designated function exit point. We successfully apply TOSS to customize Mosquito, which relies on MQTT protocol for lightweight IoT communications.

The main contributions of our work are as follows:

- We propose TOSS, an automated framework for customizing online servers and software systems using only binaries. Given a list of desired features, TOSS automatically identifies feature-related program code and customizes it to provide just-enough features to support desired services in accordance with user needs.
- For effective feature identification, TOSS leverages dynamic tainting to track feature execution by tainting relevant user request packets/fields. It harnesses dynamic tracing and symbolic execution to improve code coverage and efficiently discover more execution paths relevant to the desired features.
- Evaluation using real-world applications such as Mosquito shows that TOSS can efficiently customize software binaries, generating debloated program binaries, and eliminate potential vulnerable code without requiring any knowledge of the exploits.

2 MOTIVATION

In this section we present a statement of our feature customization problem, formally define it, and outline an overview of our TOSS approach.

**Background:** Excessive program features/code often result in increased amount of vulnerabilities. It has been shown that there are on average 15 to 50 errors per 1000 lines of industry-level code [9]. Feature customization creates customized software systems that contain just-enough features to support only desired services/utilities and can considerably reduce the software’s attack surface, as unnecessary and unwanted features are eliminated even before zero-day exploits.

As mentioned in section 1, the main challenges of TOSS is to customize online servers and software systems with only program binaries available, e.g., in legacy systems. Previous work on binary reuse can also be considered as a form of customization, even though binaries extracted and reconstructed from execution traces are only guaranteed to accept the inputs used to generate the execution traces. As a result, the reconstructed program often can only run with extremely limited inputs with exactly replicating the program behaviors. While multiple traces can be merged to increase the adaptability, it is very difficult to achieve sufficient code coverage and provide a customized binary that works with all the reasonable inputs [6].

Instead of extracting reusable parts of the binary, TOSS rewrites the static binary by keeping the necessary features while erasing others. It comprises two tasks: (i) identifying program features from a binary executable by analyzing dynamic traces that invoke different features, and (ii) rewriting the binary, in accordance with user needs, to create customized, self-contained programs.

To introduce our problem of software customization, we first need a definition of what a feature is in binary code.

**Definition 1. Feature.** A program feature is defined as a set of basic blocks - denoted by $F_i = \{f_1^i, f_2^i, ..., f_n^i\} \subseteq \mathcal{F}$ - which uniquely represent an independent, well-contained operation, utility, or capability of the program. A feature at the binary level may not always correspond to a software module at the source level. We use $\mathcal{T} = \{F_i, \forall i\}$ to denote the set of all available features in the program.

**Problem Statement:** The goal of TOSS is that, given a program binary, test cases invoking different program features, and user’s customization requirement (i.e., a set of desire features $\mathcal{T}' \subseteq \mathcal{T}$), it will produce a modified binary that contains the minimum set of functions to satisfy the user’s customization requirement and to support all desired features in $\mathcal{T}'$.

**Scope:** We focus on the customization of online servers and software systems, which are often implemented using a client-server architecture based on the underlying network protocols, such as MQTT [7]. In this paper, we assume that only program binary is available for customization. If some specific inputs are needed to execute a feature, we assume we are provided with such test-cases to execute the program with the desired feature.
3 SYSTEM DESIGN

TOSS consists of two major modules: feature identification and feature rewriting. Its system architecture is illustrated in Figure 1. Users provide customization requirement (i.e., a list of features that are needed) as well as test-cases to reach different features. TOSS takes the program binary and customization requirement as inputs and generate a customized binary consisting of only the desired features.

The feature identification module is explained in 3.1. Through program tracing and tainting-based symbolic execution (TSE), TOSS is able to find the basic blocks that are necessary to perform the desired features. Feature identification module will output the instruction addresses that are related to the target feature to the next module, feature rewriting.

The feature rewriting module is explained in section 3.2. It modifies the program binary in accordance with user’s customization requirements. The instructions remaining in the customized program can be viewed as a subset of that of the original program (only exception is the handling of function exit), which is able to retain the behavior of only the desired features. Finally, to evaluate and improve the soundness of our customization, We perform fuzzing to check the feasibility and code coverage of the customized program. We separate the fuzzing inputs into two categories: 1) benign inputs that belong to the desired features and should be processed exactly as they are handled by the original program; 2) malicious inputs that do not belong to the desired features and should not be processed. Once benign inputs cause the program to malfunction, this input will be taken to the previous feature identification module and generate a new execution trace, from which more code segments related to the desired features are discovered and added to the customized binary.

3.1 Feature Identification

As a feature-oriented customization framework, TOSS needs to discover basic blocks that are related to the target features. Previous work for feature customization, e.g., [5], often requires seed functions to bootstrap the feature identification process. Each seed function uniquely defines a feature and can be used as a source of slicing techniques that will find other functions belonging to the same feature. However, such seed functions are hard to obtain in practice. Users and even administrators often do not have detailed information regarding the implementation of various program functions.

In TOSS, we focus on online servers that are implemented using a client-server architecture. It is natural to consider network packets as the starting point of feature identification. In particular, TOSS’s feature identification module performs program execution tracing, tainting and symbolic execution to extract the code that is necessary for the desired features. The traffic among target hosts usually consists of packets that have different formats. We distinguish packets according to their formats and select the packets that need to be targeted for customization. After symbolic execution is applied without tainting information, redundant paths will be explored other than those starting from node 1. The gray nodes contain tainted instructions, which can help guide the symbolic execution to improve the effectiveness of path exploration.

Take the control flow in figure 2 as an example, where a packet is received and processed. First the format of the packet is checked. Based on what type of packet it is, basic block 2 or 6 will be invoked. The solid arrow indicates the actual execution path during the program execution, which is 1->2->3->4->6. Without symbolic execution, only this single path is considered as the code related to this packet. After customization, the new binary will not be able to process the packets that will lead to node 6. The execution paths marked with dashed arrows are discovered by symbolic execution.

On the other hand, if symbolic execution is applied without the tainting information, redundant paths will be explored other than those starting from node 1. The gray nodes contain tainted instructions, which can help guide the symbolic execution to improve the effectiveness of path exploration.

The details will be explained later.

3.1.1 Instruction Tracing. TOSS obtains runtime information through a whole system emulator TEMU [15]. We dump register values, instructions and their memory addresses while the program executes on the guest OS in TEMU. When the program is launched, every instruction gets executed will be logged. The instructions
get and propagate the values from the memory location (where the network packets are stored) will be tainted.

3.1.2 Tainting-based Symbolic Execution. Since instruction tracing can only discover the code that gets executed in specific program runs, the code coverage of target features is relatively low. To this end, TOSS uses symbolic execution to further explore the code that are related to the same type of packets (packets with the same format).

Symbolic execution is usually resource-intensive in terms of memory and CPU circles. In TOSS, we use tainting-based symbolic execution (TSE) to limit the number of memory locations/variables that need to be made symbolic, and apply extra conditions to limit the value range of certain packet fields.

- TOSS performs concolic execution in TSE module. It snapshots the value of registers at the point of tainting started, only symbolizes the variables that are related to the target packets or fields, and keeps other concrete values.
- Based on the packet format information which we know in advance, there may exist fields that have limited range of value, in which case we can apply this range constraints to the symbolic variables while performing symbolic execution. In particular, such constraints could include the option/flag of a program, packet field length, packet field maximum/minimum range, etc.

As shown in figure 3, where a connect request/packet from MQTT [7] subscriber to broker is captured during (monitored) execution, then certain fields are selected and cast into symbolic values. In general, we assume that the packet format information is available to us so that we can identify the fields whose values have meaningful variations and can lead to different execution paths of the target features.

3.1.3 Merging. Instruction tracing and TSE are complementary and together provide a faster and more efficient design for feature discovery and code coverage. On one hand, dynamic tracing can precisely locate the basic blocks that get executed. With tainting enabled, it is also able to mark the places that are related to registers/memory locations of interest. Nonetheless, it is obvious that one iteration of tracing can only get the code that processes specific packets. There are potential branches also related to the same feature but not chosen in the particular executions. To this end, symbolic execution is able to explore more branches that are not selected in specific program runs. However, without properly trimming the searching space, it will easily dive into the path explosion problem. This is where the tainted locations from instruction tracing can serve as indicators to guide and accelerate symbolic execution.

TOSS combines the addresses obtained from runtime instruction tracing and TSE to build a library of addresses that will be kept when performing binary rewriting to customize the program binary. As shown in figure 2, all basic blocks from 1 to 7 can be discovered by feature identification module, with 1 to 4 being discovered through program tracing and the rest from TSE.

If multiple types of packets that cannot be generated in one run are given as the inputs to feature identification module, TOSS will perform the above operations one by one and then merge the basic blocks discovered from multiple iterations.

3.2 Feature Rewriting

Feature rewriting creates a customized binary that consists of the desired, feature-related code segments discovered through feature identification. This section describes the three main steps that TOSS performs for feature rewriting.

Address Processing

Since a single execution trace and symbolic execution may not reach all desired program features, it requires us to first merge multiple outputs from feature identification.

TOSS will collect traces from different program executions to identify and compute the union of the related feature-constituent functions. Let \( \bar{F} \) be a set of target program features for rewriting. If the constituent basic blocks of each feature \( F_i \in \bar{F} \) can be successfully identified, we can simply create a superset of their constituent basic blocks, i.e., \( \hat{F} = \bigcup F_i \). Binary rewriting techniques are developed next to create a customized program by retaining only the features in \( \hat{F} \).

In order to rewrite the binary, we need to obtain the static addresses of instructions. We locate the static instructions based on their offsets from the entry point (starting address of “.text” section). The offsets can be calculated by subtracting the actual runtime segment address of “.text” from the runtime instruction address. The static addresses are then passed to binary rewriter as inputs to customize the instructions.

Binary Rewriting

We adopt basic block level binary rewriting in TOSS. In particular, we use DynInst, a static binary rewriter to modify the program binary. The PatchAPI in DynInst abstracts the program basic blocks in the form of CFG, which is compatible with our framework. In the case of feature removal, the goals of binary rewriting are twofold:

- The basic blocks to be eliminated should not be called. The call site of the eliminated code will be replaced to redirect the program to exit point;
- In case the undesired code get called through malicious operations/attacks such as Return-Oriented Programming (ROP), we replace the rest of the undesired basic blocks with “NOP” (except for the shared code and data segments).

The solution is illustrated in Figure 4. The original control flow is from basic block \( B_1 \) to \( B_3 \) via a “call” instruction as arrow 2 indicates. If \( B_2 \) is the target to be removed, TOSS will change the call site in \( B_1 \) and redirect it to \( B_3 \) (an exit point) as Arrow 1 indicates. In
We implement a prototype of TOSS using several binary analysis tools. In particular, the PatchAPI is used to instrument and modify the binary. The PatchAPI abstracts the program into CFG and most of the rewriting operations are performed upon it. The CFG abstraction includes functions, basic blocks and control flows. Our implementation (i) removes target features by removing the corresponding basic blocks from the original program binary, discovered by program tracing and by Tainting-based Symbolic Execution; (ii) redirects the jumps to removed basic blocks toward program exit point.

5 EVALUATION

In this section, we evaluate the performance of TOSS and the effect of feature customization.

Experiment Setup: Our experiments are conducted on a 2.80 GHz Intel Xeon(R) CPU E5-2680 20-core server with 16 GByte of main memory. The operating system is Ubuntu 14.04 LTS. We perform feature identification and rewriting on Mosquito [7]. The features we choose to keep in the customized binary are “topic” and “message” in the Mosquito_pub.

As shown in table 1, we collect the number of instructions that are in the original program binary, discovered by program tracing and by Tainting-based Symbolic Execution.

Case Study: MQTT Message Queuing Telemetry Transport (MQTT) is a protocol using a list of topics to subscribe and publish messages, widely used in Internet of Things (IoT) applications. There are three basic entities in MQTT protocol, e.g., a broker, a publisher and a subscriber. The subscriber signs up for topics via the broker, who receives updates/messages from the publisher and then delivers them to subscribers according to the selected topics. MQTT packets typically contain three fields, i.e., fixed header, variable length header, and payload. The fixed header consists of control header and packet length fields. The variable length header is used to enable extra features. In this paper, we perform feature customization on Mosquitto 1.5, a lightweight implementation of MQTT.

We analyze the protocol functionalities and select the desired features to keep for a customized version, which maintains basic MQTT communications among publisher and subscriber, while eliminating other unnecessary features. For Mosquito publisher, we identify hostname, port number, topic and message as features necessary to publish messages in MQTT protocol. For Mosquito subscriber, we identify hostname, port number and topic as features necessary to subscribe or receive messages/updates.

By keeping only the necessary features mentioned above, the customized binary have the following benefits:

- The program binary is lighter-weight and has better performance (as “NOP”’s will be optimized by the system during execution at runtime).
• The program binary eliminates many redundant features and significantly reduces the corresponding attack surface.

Among those removed features, some are particularly security-related. For example, the option “insecure” is a feature for subscriber and publisher to establish unencrypted and unauthenticated connection. When it is enabled, authentication of server hostname will be skipped, meaning that a malicious third party could gain the access to the MQTT communication. Another feature “publish file” allows a publisher to send files through an MQTT connection. This often results high risks due to potential buffer overflow or malicious code injection, as evidenced by the vulnerabilities found in Apache Struts, leading to the Equifax data breach. The feature “will” can enable broker to cache/save a message with certain payload, QoS, retain and topic. When the client disconnects unexpectedly, the message will be automatic sent from either subscriber or publisher. This may cause information leakage or other unexpected behaviors especially when a “will” request comes from the publisher.

As shown in table 1, the customized Mosquitto publisher program binary only contains 1124\+188=1312 instructions and can still fully support the desired feature, whereas the original program binary in comparison contains 3305 instructions.

Case Study: Computation Offloading (CO) server. This is a light-weight server and client module that implements the computation offloading functionality, a popular computing paradigm for mobile devices. A client can send task parameters (as input) to a server, who performs the required computation and deliver the results to the client. Performing the computation at the server side enables energy/latency reduction for the client device. In our CO implementation, the packets sent from the client to the server contain a header which specify the request type and packet length, while the payload includes the value of an operator and corresponding operands. When the server receives the request, a packet format checker will verify the validity of the payload. Suppose only one operation, “Addition”, is needed, we can remove the other types of computations (i.e., unnecessary features) in the server, while preserving format checking and addition features. A maximum number of operands of “Addition” is defined within the security policy as “Addition Operand Max”. In specific executions, the packets may not exceed this limitation. However, the code at server side that handles this exception will have to be kept in the customized problem, which is achieved by our proposed TSE that has the ability to explore different branches that are not taken in specific executions.

6 DISCUSSION
As a work-in-progress, TOSS has the following limitations which will be considered as future work:

Lack of backward tainting: The tainting module in feature identification phase currently can only deal with forward tainting but not backward tainting, e.g., only the instructions that process inbound packets can be tainted, with the fields in inbound packets properly marked (as the source of tainting). If the taint source is available in the outbound packet, then to taint the instructions that generated such packets requires backward tainting. We consider it as a future work to add backward tainting capability in our TOSS framework.

Lack of support for obfuscated binary: As our rewriting module performs rewriting on static binary, it requires the precise instruction addresses for rewriting. Obfuscated binary cannot be supported in this work.

7 RELATED WORK
De-bloating: De-bloating has been studied to analyze and mitigate program bloat caused by feature creep [3, 3–5, 11, 20]. At source code level, Yufei Jiang et al. perform program slicing and data analysis to remove code segments that are related to the target feature [5]. In particular, they discover and delete code that has dependencies with its return value, parameter and call site. Some knowledge of feature-related function (seed function) are required to bootstrap the slicing. Jred [4] aims to remove unused methods in JAVA program and libraries by analyzing the program call graph. It operates at IR level, i.e., the JAVA bytecode is lifted into Soot IR. After trimming, IR is transformed into Java bytecode to produce a light-weight program. While the goal of above works is to remove redundant code from program, we offer more flexible ways to customize the (combination of) program features. Moreover, most of the previous de-bloating techniques can only work with object-oriented programming languages while TOSS can be directly applied to binaries.

Binary reuse: Binary reuse has been addressed by several works [18, 23, 25]. The reuse of binary code, different from source code, carries great difficulty. Methods proposed in [2] identify self-contained code fragment from binary with the help of both static disassembling and dynamic execution monitoring. There are also research works that focus on reconstructing program binary from dynamic traces, by utilizing instruction trace and memory dump [6]. However, neither of the above two methods fits in the context of program feature customization due to limited degree of flexible modification, as it only focuses on segment reuse and high level assembly code. Moreover, even if the code can be customized, the newly compiled binary may fail to fulfill the purpose of feature customization.

Symbolic execution and fuzzing: In this paper we leverage the tainting information to guide the symbolic execution. Existing works have studied multiple methods on a more effective symbolic execution. Driller [17] interactively applied fuzzing and symbolic execution to explore code based on the observation that fuzz-generated inputs often fail to pass the input checking (while symbolic execution can easily generate the such conditions) and symbolic execution can easily dive into the issue of path explosion (while fuzzing is a relatively light-weight technique to explore code within certain scope). Directed greedybox fuzzing leverages annealing-based heuristic to generate inputs that can reach a certain point in the program, achieving a better performance over directed white-box fuzzing and undirected gray-box fuzzing. Hang Zhang et al. [24] try to verify the presence of program patches in program binary. They adopt symbolic execution to extract semantic formulas from the binary which are then compared against the code signature discovered from patches in source code. StraightTaint [10] also combine tainting (“incomplete” taint propagation) and symbolic execution to improve the runtime tainting performance (by
lightweight analysis) while still keep necessary information for offline analysis. StatSym [21] employs runtime predicates to guide the symbolic execution. It collects and analyzes certain program states from the sampling execution (benign and buggy) then use them to guide symbolic execution engine to explore the most possible places where a bug could happen.

**Binary analysis tools:** A chain of binary tools have been widely used to analyze binary code for different purposes, such as binary CFG analysis, vulnerability detection and binary rewriting. Specifically, binary rewriting tools such as DynInst [16] and Pin [8] are able to perform binary modification either statically or dynamically. In this paper, we employ DynInst to perform basic-block modification of program features. In feature identification module, we use TEMU [15] to emulate a system where the web servers are launched then monitored. TEMU also contains a tainting plugin ("tracecap") that can taint the instructions from network packets. Angr [14] is used to perform symbolic execution together with the tainting information obtained through tracecap.

### 8 Conclusion

We design and evaluate a binary customization framework TOSS for online server systems. TOSS aims to generate customized program binaries with just-enough features and can satisfy a broad array of customization demands. Feature identification and feature rewriting are two major modules in TOSS, with the former discovering target features and related code using program tracing and tainting-based symbolic execution, and the latter modifying the program binary to reconstruct a customized program with only desired features. Our experiment results demonstrate that TOSS is able to effectively keep code segments related to the target features and erase undesired code, e.g., keep the necessary functionalities and reduce the attack surface.

### References


