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 Mosquitto, 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.

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 leads 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 depends 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 feature. 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 feature. 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 the 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 Mosquitto, 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 Mosquitto shows that TOSS can efficiently customize software binaries, generating debloated program binaries, and eliminate potential vulenrable 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, eventhough 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

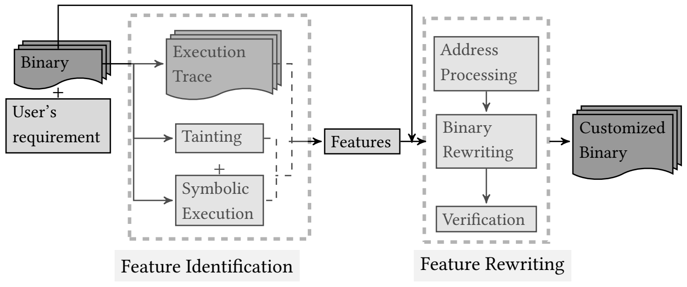


Figure 1: TOSS System Diagram

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 trace that invokes 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 $F_i = \{f_i^1, f_i^2, ..., f_i^n\} \subseteq \mathcal{F}$ – 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.

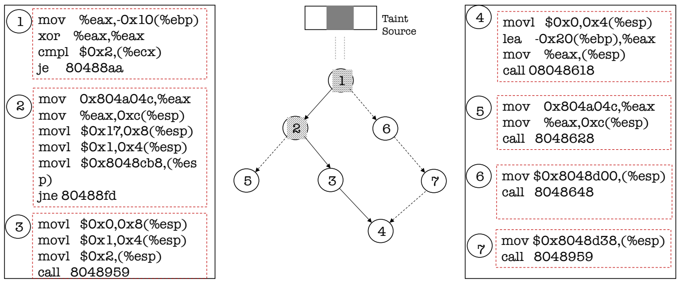
Problem Statement: The goal of TOSS is that, given a program binary, test cases invoking different program features, and user's customization requirement (i.e., a set of desire features $\hat{\mathcal{T}} \subseteq \mathcal{T}$), it will produce a modified binary that contains the minimum set of functions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

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



```
1  mov   %eax,-0x10(%ebp)
   xor   %eax,%eax
   cmpl  $0x2,(%ecx)
   je    80488aa

2  mov   0x804a04c,%eax
   mov   %eax,0xc(%esp)
   movl  $0x17,0x8(%esp)
   movl  $0x1,0x4(%esp)
   movl  $0x8048cb8,(%es
   p)
   jne 80488fd

3  movl  $0x0,0x8(%esp)
   movl  $0x1,0x4(%esp)
   movl  $0x2,(%esp)
   call  8048959
```

```
4  movl  $0x0,0x4(%esp)
   lea   -0x20(%ebp),%eax
   mov   %eax,(%esp)
   call 08048618

5  mov   0x804a04c,%eax
   mov   %eax,0xc(%esp)
   call  8048628

6  mov $0x8048d00,(%esp)
   call  8048648

7  mov $0x8048d38,(%esp)
   call  8048959
```

Figure 2: Feature identification by combining instruction tracing and tainting-based symbolic execution

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. 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 remaining features and should be processed as if they are given to the original program; 2) malicious inputs that don't belong to the remaining feature and shouldn't 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 target feature 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 features to keep. 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 remain after customization. (Different formats can be triggered by user-defined options or environment configurations). When the program is executed in a system emulator and the target packets are detected, the related basic blocks will be tainted and logged.

TOSS employs two approaches to collect the feature-related basic blocks: (1) logging the execution traces when the program starts on the emulated system; (2) tainting the execution intructions based on the target packets and perfrom symbolic execution to discover more branches.

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. 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 can help guide the symbolic exectuion to improve the effectness 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 runs 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 networ packets are stored) will be tainted.

3.1.2 Tainting-based Symbolic Execution. Since instruction tracing can only discover the code that get executed in specific program runs, the code coverage for the target feature is relatively low. 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-consuming in terms of memory and CPU circles. In TOSS , we use tainting-based symbolic execution (**TSE**) to limit the number of locations that are 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 fileds, and keeps other concrete values.
- based on the packet format information which we know aforehand, 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 can be the option/flag of a program, packet field length, packet field mamimum value, etc.

As shown in figure 3, where a connect packet from MQTT [7] subscriber to broker is captured during monitored execution then certain fields are chosen as symbolic value. In general, we assume that the packet format information is available to us so that we can

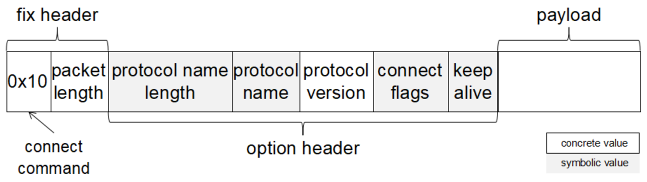


Figure 3: Select packet fields to symbolize

identify the fields whose values have meaningful variations and can lead to different execution paths.

3.1.3 Merging. Instrcution tracing and TSE are complementary and together provide a faster and more efficient design for code discovery. On one hand, dynamic tracing can precisely locate the basic blocks that get executed. With tainting enabled, it's also able to mark the places that are related to interested registers/memory locations. Nonetheless, it's obvious that one iteration of tracing can only get the code that processes specific network packets. There are potential branches also related to the same type of operation but not taken in particular runs. On the other hand, symbolic execution is able to explore more branches that are not taken 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 indications to improve and accelarate 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 each iteration.

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 $\hat{\mathcal{F}}$ be a set of target program features for rewriting. If the constituent basic blocks of each feature $F_i \in \hat{\mathcal{F}}$ can be successfully identified, we can simply create a superset of their constituent basic blocks, i.e., $\hat{F} = \cup F_i$. Binary rewriting techniques are developed next to create a customized program by retaining only the features in $\hat{F}$.

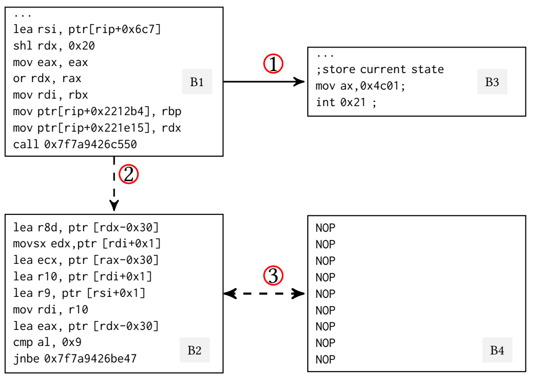

```
...
lea rsi, ptr[rip+0x6c7]
shl rdx, 0x20
mov eax, eax
or rdx, rax
mov rdi, rbx
mov ptr[rip+0x2212b4], rbp
mov ptr[rip+0x221e15], rdx
call 0x7f7a9426c550
```
```
...
;store current state
mov ax,0x4c01;
int 0x21 ;
```
```
lea r8d, ptr [rdx-0x30]
movsx edx,ptr [rdi+0x1]
lea ecx, ptr [rax-0x30]
lea r10, ptr [rdi+0x1]
lea r9, ptr [rsi+0x1]
mov rdi, r10
lea eax, ptr [rdx-0x30]
cmp al, 0x9
jnbe 0x7f7a9426be47
```
```
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
```

Figure 4: An illustrative example of implementing feature rewriting on OpenSSL.

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 substract the actual runtime segment address of ".text" from the runtime instruction address. The static addresses are then passed to binary rewriter as inputs to modify 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_2 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 addition to the change of control flow, TOSS will also replace B_2 with "NOP"s to prevent invoking the removed features at runtime, e.g., through ROP [13].

Verification

After binary rewriting, standard program fuzzing techniques [22] can be employed by TOSS to validate the effectiveness and correctness of feature rewriting. A fuzzing engine generates test cases that can be categorized into two sets: $\mathcal{D}$ that invoke the desired features in customized program, and $\mathcal{E}$ that involve at least one of the eliminated features. In particular, TOSS uses $\mathcal{D}$ to confirm the integrity of necessary program functionalities, while $\mathcal{E}$ helps verify the successful removal and handling of eliminated features. We note that this validation procedure can also be performed using user/admin provided test cases. If a benign input (belong

Table 1: Number of instructions discovered by tracing and TSE on Mosquitto_pub

Instructions	Mosquitto_pub	CO_server
original binary	3305	509
from tracing	1124	332
from TSE	188	37
customized binary	1312	369
Features removed	insecure, publish file	computation except addition

to the features remaining in the binary and should be processed) causes the program to crash, we will take this input to generate the corresponding execution trace, and discover the instructions in the execution trace to complement the current binary. This step of verification serves as a feedback to the feature identification module.

4 IMPLEMENTATION

We implement a prototype of TOSS using several binary analysis tools.

Tracing and tainting: We implement our tracing and tainting module in TEMU. To enable finer-grained tainting such as field tainting, we instrument the plugin "tracecap" in TEMU to add filters when setting the taint bitmap, e.g., to enable tainting for packets with certain formats or for certain fields in the packets.

Symbolic execution: We use Angr [14] to perform symbolic execution on the static binary. The symbolic execution starts from the places where packets are stored, which can be identified by the tainting information. Through the symbolic execution, we dump the addresses of basic blocks that have been explored.

Binary rewriting: We use DynInst for static binary rewriting. In particular, the PatchAPI is used to instrument and modify the binary. 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 CFG list and replacing the basic block body with *NOP*s; (ii)redirects the jumps to removed basic blocks to 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 Mosquitto [7]. The features we choose to keep in the customized binary are "topic" and "message" in the mosquitto_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 Exection.

Case Study: MQTT Message Queuing Telemetry Transport (MQTT) is a protocol using a certain topic to subscribe and publish message which is always used in internet of things (IOT). There are three entities during MQTT communication, e.g., broker, publisher and subscriber. The subscriber subscribes a topic via broker before it can receive messages published by publisher (via broker). The MQTT packets contain three fields which are fix header, variable length header and payload. The fix header consists of control header and packet length. The variable length header is used when some extra features are enabled. In this paper, we perform feature customizations on Msoquitto 1.5, a lightweight implementation of MQTT.

We analyze the protocol functionalities and select the necessary features to keep for a basic version that can maintain the MQTT communications among publisher and subscriber. For Mosquitto publisher, hostname, port number, topic and message are features to publish messages in MQTT communication. For Mosquitto subscriber, hostname, port number and topic are features to subscribe or receive messages in MQTT communication.

By keep only the necessary feature mentioned above, the customized binary have the following benefits:

- the program binary is lighter-weight and has better performance("NOP"s will be optimized by the system during runtime and consumes much less resources).
- the program binary erases most of the redundant features and reduce the attacking surface.

Among those removed features, some are particularly security-related. The option "insecure" is a feature for both subscriber and publisher. When it's enabled, verification of the server hostname will be skipped which means a malicious third party could gain the access to the MQTT communication. The feature "publish file" is a feature that sents files through MQTT connection from the publisher. File transmissions are sometimes vulnerable such as the vulnerability in Apache Struts, which causes the Equifax data breach. The feathre "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 the "will" comes from the publisher.

As shown in table 1, the customized mosquitto publisher program binary only contains 1124+188=1312 instructions and can still work with the remaining feature while the original program binary contains 3305 instructions.

Case Study: Computation Offloading (CO) server This is a light-weight server and client model that implements the computation offloading functionality, which is a common computation scheme in mobile devices. The client can send computation parameters to the server and get the results from server. The actual computation is performed at the server side to archieve energy/latency reduction on client side. 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 include the value of operator and operands. When the server receives the request, packet format checking will be performed before it starts to process the payload. Suppose only the operation "Addition" is needed, we can remove other types of computation in the server side after format checking. A maximum number of operands of "Addition" is defined within the security policy as "Addition_Operand_Max". In particular runs, the packets may not exceed this limitation. However, the code segment in server side that handle this exception will have to be kept in the customized version, where TSE can help to explore the branches that are not taken.

6 DISCUSSION

As a work-in-progress, TOSS has the following limitations which are considered as future work:

Lack of backward tainting: The tainting module in the feature identification phase currently can only deal with forward tainting but not backward tainting, e.g., only the instructions that process the inbound packets can be possibly tainted, given that the inbound packets are marked as the source of tainting. If the taint source is the outbound packet, then the instructions that generate such a packet cannot be marked as tainted. To support tainting on outbound packet, backward tainting module can be added to the tainting implementation in TEMU.

Lack of support for obfuscated binary: As our rewriting module performs rewriting on static binary, which require the precise addresses for instruction 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 fuzzer-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 greybox fuzzing leverages annealing-based heuristic to generate inputs that can reach a certain point in the program, archieving 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 propogation) and symbolic execution to improve the runtime tainting performance (by lightweight logging) 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 in this paper. 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 one discovering the target features using program tracing and tainting-based symbolic execution, and the latter one modifying the features to reconstruct a customized program. 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

[1] Tiffany Bao, Johnathon Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. Byteweight: Learning to recognize functions in binary code. USENIX.

[2] Juan Caballero, Noah M Johnson, Stephen McCamant, and Dawn Song. 2009. *Binary code extraction and interface identification for security applications*. Technical Report. CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE.

[3] Yurong Chen, Tian Lan, and Guru Venkataramani. 2017. DamGate: Dynamic Adaptive Multi-feature Gating in Program Binaries. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*. ACM, 23–29.

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

[5] Yufei Jiang, Can Zhang, Dinghao Wu, and Peng Liu. 2016. Feature-based software customization: Preliminary analysis, formalization, and methods. In *High Assurance Systems Engineering (HASE), 2016 IEEE 17th International Symposium on*. IEEE, 122–131.

[6] Yonghwi Kwon, Weihang Wang, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. 2017. CPR: cross platform binary code reuse via platform independent trace program. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 158–169.

[7] Roger A Light. 2017. Mosquitto: server and client implementation of the MQTT protocol. *Journal of Open Source Software* 2, 13 (2017).

[8] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, Vol. 40. ACM, 190–200.

[9] Dan Mayer. 2012. Ratio of bugs per line of code. (2012). https://www.mayerdan.com/ruby/2012/11/11/bugs-per-line-of-code-ratio

[10] Jiang Ming, Dinghao Wu, Jun Wang, Gaoyao Xiao, and Peng Liu. 2016. StraightTaint: Decoupled offline symbolic taint analysis. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, 308–319.

[11] Nick Mitchell and Gary Sevitsky. 2007. The causes of bloat, the limits of health. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 245–260.

[12] Gail C Murphy, Albert Lai, Robert J Walker, and Martin P Robillard. 2001. Separating features in source code: An exploratory study. In *Proceedings of the 23rd international Conference on Software Engineering*. IEEE Computer Society, 275–284.

[13] Marco Prandini and Marco Ramilli. 2012. Return-oriented programming. *IEEE Security & Privacy* 10, 6 (2012), 84–87.

[14] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 138–157.

[15] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A new approach to computer security via binary analysis. In *International Conference on Information Systems Security*. Springer, 1–25.

[16] Open Source. 2016. Dyninst: An application program interface (api) for runtime code generation. *Online, http://www. dyninst. org* (2016).

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

[18] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 934–953.

[19] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2010. Finding low-utility data structures. *ACM Sigplan Notices* 45, 6 (2010), 174–186.

[20] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, and Gary Sevitsky. 2010. Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 421–426.

[21] Fan Yao, Yongbo Li, Yurong Chen, Hongfa Xue, Tian Lan, and Guru Venkataramani. 2017. Statsym: vulnerable path discovery through statistics-guided symbolic execution. In *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on*. IEEE, 109–120.

[22] Michal Zalewski. 2007. American Fuzzy Lop. (2007).

[23] Junyuan Zeng, Yangchun Fu, Kenneth A Miller, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2013. Obfuscation resilient binary code reuse through trace-oriented programming. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 487–498.

[24] Hang Zhang and Zhiyun Qian. 2018. Precise and Accurate Patch Presence Test for Binaries. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD. https://www.usenix.org/conference/usenixsecurity18/presentation/zhang-hang

[25] Peng Zhang, Jianhui Li, Alex Skaletsky, and Orna Etzion. 2009. Apparatus, system, and method of dynamic binary translation with translation reuse. (Nov. 24 2009). US Patent 7,624,384.