CustomPro: Network Protocol Customization through Cross-host Feature Analysis

Yurong Chen, Tian Lan, and Guru Venkataramani

George Washington University
{gabrielchen,tlan,guruv}@gwu.edu

Abstract. The implementations of network protocols are often "bloated" due to the need to satisfy diverse user requirements and to suit different application environments. The continual expansion of program features contribute to not only growing complexity but also increased the attack surface, making the maintenance of network protocol security very challenging. Existing work either de-bloat programs at source code level (which may not always be available, in particular for legacy systems) or customize binaries only with respect to a very limited set of inputs. In this paper, we propose CustomPro , a new approach for automated customization of network protocols. We harness program execution tracing, tainting and guided symbolic execution to identify relevant code from the original program binary, and leverage static binary rewriting techniques to create a customized program binary that only contains the desired functionalities. We implement a prototype of CustomPro and evaluate its feasibility using OpenSSL (a widely used SSL implementation) and Mosquitto (an IoT messaging protocol implementation). The results show that CustomPro is able to create functional program binaries with only desired features and significantly reduce the potential attack surface by targeting and eliminating unwanted protocol features.

Keywords: Program Customization · Binary Rewriting · Cross-host tainting.

1 Introduction

Recently, network protocols have frequently become targets of cyber attacks. Even protocols that are carefully designed to enhance the security of communications (such as OpenSSL) can be exploited and leveraged, posing severe threats (such as information leakage and DoS attacks) to online users [2, 10]. Network protocols are vulnerable due to a number of reasons: a) To satisfy diverse end-user requirements and to suit different application environments, network 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 [14]. b) Even standardized protocols may have a variety of different implementations and specifications, in accordance with heterogeneous system/user

requirements (especially in IoT systems). Such inconsistency weighs on the feature creep issue and makes the management of protocol implementations much more difficult and the network connections prone to attacks. Recent 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 corresponding attack surface. Prior work on static software debloating [14, 13] 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 [44, 33, 45, 3, 18] can only achieve correct execution under very limited scope of user inputs.

In this paper, we propose a new approach, CustomPro , for automated customization of network protocols. We define automated feature customization as the process of identifying and rewriting different program features from a binary executable. Without requiring any knowledge of potential exploits, the customized programs contain just enough software features to support only the necessary functionalities, thus significantly reducing attack surface and exposure to future exploitations that leverage excessive features (e.g., zero-day attacks). Our approach goes beyond existing work on feature separation [25], reduction [14] and code de-bloating [38, 13, 37], which focus on removing unused code. We argue that vigilantly managing and customizing permitted features is crucial for achieving improved software security [14], especially for network protocols that are frequently targeted by attackers because of the value of their data and services.

At the core of protocol customization approach, a key problem is to identify software features directly from a program's binary without access to source code or debug information. We define a *feature* as a collection of basic blocks, which uniquely represent an independent, well-contained capability of the program. First, in order to probe for the basic information about the program binary and feature-related execution (such as executed instructions, values of operands or pointed memory locations), we run the testing protocol systems inside emulators. Second, since protocol features embedded in an online software system are often accessed remotedly via the network packets, triggered by varied requests and responses, we employ dynamic tainting to track the information flow in a

monitored execution of a target feature, utilizing the network packets or relevant packet fields as taint sources. Third, to enable taint proporgation in the network (which is crucial for network protocols [43, 28, 46]), we equip the tainting module with the ability of transmitting and processing taint information across different hosts. Fourth, since dynamic analysis of protocol features depends upon specific input values and executions, a pure dynamic tainting approach may be difficult to achieve sufficient code coverage, or 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 tainting and symbolic execution to compensate for the incompleteness of dynamic analysis*. We utilize dynamic tainting to guide an symbolic execution engine to effectively perform a light-weight search for any additional code blocks that belong to the target features but did not get executed in the test runs. As a result, CustomPro delivers a customized program that has the unique ability to correctly execute desired features with a wide range of user inputs.

Identifying the feature-related code segments enables us to rewrite program features based on user needs. We leverage binary rewriting tools (DynInst [31]) to obtain a customized binary that only retains the desired program features while removing the other unwanted ones. In particular, we replace undesired basic blocks with NOP instructions 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 CustomPro to customize real-world protocol implementations such as OpenSSL(a widely used SSL implementation) and Mosquitto(a popular implementation of MQTT protocol for lightweight IoT communications), by customizing the insecure or unnecessary features such as *heartbeat* (in OpenSSL) and *will* (in MQTT).

The main contributions of our work are as follows:

- We propose CustomPro , an automated framework for customizing network protocol implementations using only program binaries. Given a list of desired features, CustomPro 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, CustomPro leverages system emulation to run the test protocols in guest operating systems, from where we can get the execution information as well as data flows via a cross-host dynamic tainting. Together with symbolic execution, CustomPro further improves code coverage and efficiently discovers more executed code relevant to the desired features.
- We evaluate CustomPro using real-world applications such as OpenSSL and Mosquitto, and our results show that CustomPro can efficiently customize software binaries, generating debloated program binaries, and eliminate potential vulenrable code without requiring any knowledge of the exploits.

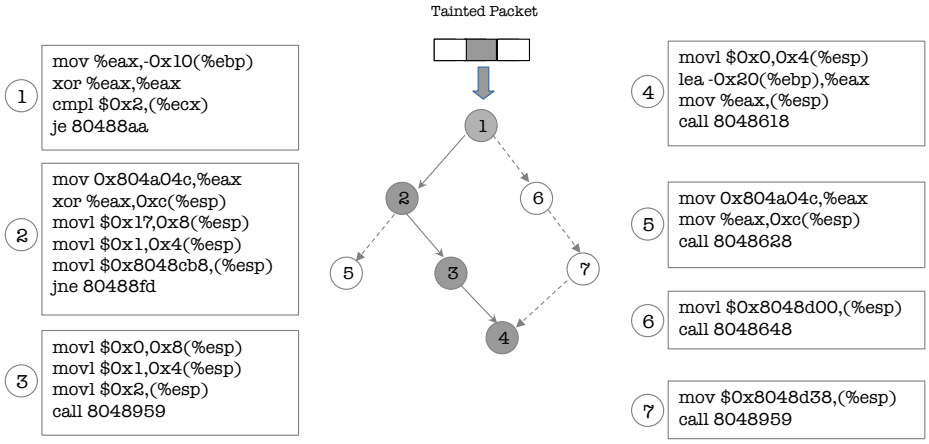

Fig. 1: Feature identification by combining tainting and symbolic execution

2 Motivation

The attacks against network systems through the vulnerabilties in protocols have never ceased and are often followed by severe consequences. *Heartbleed* [11] exposes private information such as server private keys to the Internet through a simple *Heartbeat* feature in OpenSSL. Nearly half a million certificates were exposed and over 199,500 sites are still vulnerable as of the time Jan. 2017, which is over three years after the bug was reported [16]. KRACK [34], or key reinstallation attack, a replay attack that leverages the vulnerable design in WAP2 protocol, can be exploited to gradually obtain the full keychain used for encrypting the traffic under a WIFI environment. The handshake messages are replayed by MITM (Man In The Middle) such that the end node will be deceived to reinstall the key that is already in use. The same key value will be used repeatedly for encryption, resulting in repeated occurrences of the same message. Finally, the 2017 data breach on credit reporting agency Equifax had 146.6 million people expose their names and dates of birth, and 145.5 of those expose their Social Security Numbers and/or driver's license numbers, across US, UK and Canada [12]. The attack was believed to exploit the CVE-2017-5638, which is a flaw in the Apache Strut framework. This vulnerability was discovered and reported more than two months before the data breach happened.

The key takeaways from these attacks are as follows:

- Network Protocols are a critical link in the chain of network security.
- Patching solutions can't keep up. Patches can come late while the exploits have already led to large-scale disaster. Even after a patch is released, there is no guarantee that all vulnerable entities in the network will apply the patch soon enough, leaving some of them vulnerable for a longer time.
- Hardening the security of network protocols is very different from that of single-host and offline software systems, since protocols often contain a significant amount of features and operate in a fully distributed environment.

Any effective solution must harden all of the entities collectively along with their corresponding binary code modules.

Software customization, an approach to extract desired parts and/or remove undesired ones from a target program, has been applied to reduce the attack surface and improve program security [15, 13]. In this paper, we leverage customization to 1) remove wanted program features to reduce attack surface (or the risk of being attacked), e.g., protect the program from zero-day exploits; and 2) prevent the network from being exploited via known vulnerabilities before patches becomes available. However, previous customization techniques often remove unused functions from program source code, which is not always available especially for COTS and legacy programs. To this end, CustomPro develops protocol customization that only requires only program binary for customization. Moreover, in practice it is often hard (if possible at all) to identify all functions/code related to target protocol features from only several seed functions [15]. Network protocols such as SSL handshake often implement complex state machines where different states are convoluted together and even spread across different hosts, linked by only network packets. In CustomPro , we utilize a cross-host, dynamic tainting technique to track exchanged packets and trace program executions, in order to discover the code segments related to target protocol features. CustomPro rewrites the static binary based on the user needs. It further applies symbolic execution to identify any additional code belonging to the target features that were not captured during dynamic analysis.

We utilize the example shown in figure 1 to illustrate our key idea. The feature identification starts from a tainted packet or tainted fields in a packet. The taint will propagate through basic blocks of the binary code. In this example, there are two features F_a and F_b, starting from node 2 and 6 respectively. Suppose that a user wants to keep feature F_a and remove F_b. The basic blocks represented by shaded nodes (1,2,3,4) are tainted in the current execution path, i.e., $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$, discovering code blocks (1,2,3,4). However, another execution path $1 \rightarrow 2 \rightarrow 5$ also belonging to F_a has not been identified due to limited coverage of dynamic analysis. Hence, we perform symbolic execution starting from the node 2 to explore the possibility of any other execution paths of F_a, which eventually leads to discovery node 5.

Assumptions and Scope: We assume that the packet format of a target protocol is known beforehand and the feature-related fields can be identified. In practice, packet formats can always be identified either through protocol specifications or using reverse engineering techniques described in prior work [8, 4, 20, 7]. We also assume that some (limited number of) test inputs are provided to trigger the target features and serve as starting point for our analysis. These inputs can be easily obtained from system tracing, packet sniffing, and/or fuzzing. The scope of this work is to customize a given implementation of network protocols with only access to program binaries.

3 System Overview

We formally define protocol features and problem statement as follows:

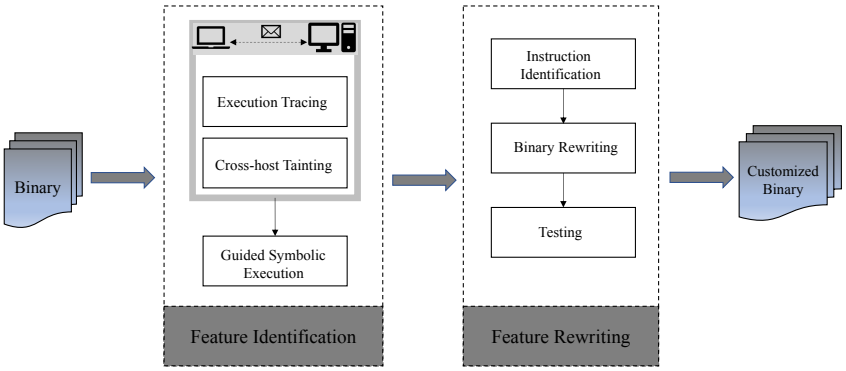

Fig. 2: CustomPro System Diagram

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 CustomPro is that, for a given program binary and 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 basic blocks/functions to satisfy the user's customization requirement and to support all desired features in $\hat{\mathcal{T}}$.

CustomPro consists of two major modules: feature identification and feature rewriting. Its system architecture is illustrated in Figure 2. Users provide customization requirement (i.e., a list of features that are needed) as well as test-cases to reach different features. CustomPro takes the program binary and customization requirement as inputs and generate a customized binary containing only the desired features.

CustomPro employs a combination of static and dynamic techniques. In particular, 1) feature identification module mainly relies on monitored program execution (inside a system emulator), dynamic analysis and symbolic execution to explore and discover the code segments related to the target features; and 2) feature rewriting module is a static binary rewriter at the core of its design. When instructions are identified by the previous module, the rewriting module keeps the desired instructions and remove others.

The feature identification module is explained in Section 4. Through program tracing, cross-host tainting and guided symbolic execution(**GSE**), CustomPro is able to find the program instructions that are necessary to perform the desired features. The discovered instructions can come from two sources: 1. Execution tracing and tainting form the basis of feature identification, in which executed program instructions are logged and those related to target features are tainted. 2. With the information above, GSE performs a light-weight search for additional code blocks that are also related to the target feature but not executed in the test

runs. The feature identification module then combines and passes the collectively identified instructions (that are related to the target features) to the feature rewriting module.

The feature rewriting module is explained in Section 5. 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.

4 Feature Identification

As a feature-oriented customization framework, CustomPro discovers basic blocks that are related to target features in program binary. Previous work for feature customization, e.g., [15], often requires seed functions to bootstrap the feature identification process. However, such seed functions are difficult to obtain in practice. Users and even administrators often do not have detailed information regarding the implementation of various protocol features/functions. CustomPro instead considers network packets as the starting point of feature identification, as it focuses on network applications and protocols. In particular, CustomPro 's feature identification module performs program execution tracing, cross-host tainting and symbolic execution to discover the relationships between features and their coresponding code.

4.1 Execution tracing

CustomPro starts with test inputs that trigger the target features, during which all code related to the target features are captured and identified dynamically. To this end, we employ dynamic program analysis to discover the code and other runtime infomation related to the target features. We run the program inside a whole system emulator-TEMU [41], where the instructions get executed will be logged and tainted instructions are labeled. The taint propagation mechanism will identify all code related to certain packets/fields. In fact, during the program execution, other runtime information such as operand values and CPU register values are also inspected and logged. These values are used later for a symbolic execution as described in section 4.3.

Network protocols typically involve executions on multiple network entities with different roles, such as servers and clients. We execute all relevant entities

on the guest OS inside TEMU, and implement a cross-host tainting mechanism to propagate taints between multiple entities. As will be described in section 4.2, we piggyback taint information onto existing network packet, which requires the modification of both sender and receiver entities.

4.2 Cross-host Packet Tainting

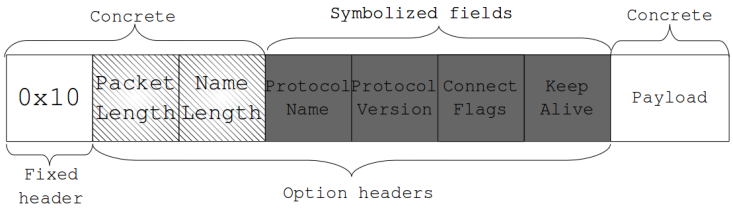

Fig. 3: Field symbolization in tainted fields

Not all of the logged instructions are related to the target features. One intuitive approach to identify relevant instructions is tainting. However, whole-packet-level tainting may fail to achieve the granularity needed to extract the code of the target feature. For example, in *ClientHello* message of SSL protocol, there could be undesired extensions such as *HeartBeats* along with other necessary fields. If the customization requirement is to remove heartbeat feature, packet-level tainting cannot distinguish *HeartBeats* from other features embeded in the same packet.

In CustomPro , we apply muti-tag, field-level tainting to label instructions according to the features they belong to. In particular, we classify tainted instructions into two sets: 1) $\mathcal{K}$: code related to desired features and will be kept in the customized binary and 2) $\mathcal{R}$: code related to undesired features and will be removed after customization. Note that untainted code will not be removed from the original binary since they are related to program initialization or state transitions. In addition, the instructions in $\mathcal{K}$ and $\mathcal{R}$ will be utilized during partial symbolic execution to identify any missing code blacks. The details will be explained in Section 4.3.

The tainting engine in TEMU maintains the taint tags in shadow memory. In our case, the shadow memory will track the taint status of every byte in NIC buffer. When the taint source and tag are specified at the packet (i.e., which fields are tainted and which tag each field gets), the corresponding memory location is tainted. The taint then will be propagated along with data flow such as read, DMA (Direct Memory Access), table lookups and arithmetic operations. By default, TEMU will taint the whole packet if it satifies the user-defined filters such as TCP packet and UDP packet. For a finer-grained tainting, we instrument the tainting engine in TEMU to enable field tainting on packets. The target fields are identified through their offsets in the packets.

In order to track the data flow across different hosts, CustomPro also implement a cross-host tainting mechanism [43] to transmit taint information in the network. This is essential for protocols that contain state machines on both server and client sides. Take SSL handshake process as an example, when the server is listening to incoming connections, it stays at a state ready to read *ClientHello*. After the *ClientHello* message is received and processed, the server will go into another state such as replying *ServerHello*, renegotiation or error state depending on the result of processing *ClientHello*. Suppose a *ServerHello* is sent to the server side, the client will change its state to reading the *ServerHello*. Such iterations extend the scope of data flow from one individual host to multiple hosts across the network, while the status of one execution depends on the executions on all other hosts in the protocol. To enable cross-host tainting, we piggyback taint information onto each packet flight. The taint information contains an offset table that indicates which bytes in the current packet are tainted and which labels are used to taint them, allowing the taints to be extracted and processed at the recipient.

4.3 Guided Symbolic Execution

By utilizing tainting, CustomPro now can map each instruction to its feature, on all participating entities of the protocol. To customize the binary, a straightforward approach would then keep the desired instructions (related to desired features) and remove the rest. However, there is still one limitation to this approach: The given test inputs can only trigger specific execution paths in the binary code, which may not provide a full coverage of the target feature execution.

To this end, after tracing and tainting, we take the execution traces and tainting information as the input to perform guided symbolic execution(**GSE**), in order to discover any additional code blocks that are related to the target features/fields. Symbolic execution is usually resource-consuming in terms of memory and CPU circles. To trim the searching space of symbolic execution, we (i) leverage the tainting results as well as the runtime information from execution tracing, to limit the number of locations that require symbolic execution and (ii) infer conditions from execution logs to further concretize certain variables as well as to limit the value ranges of certain symbolized fields. In particular, our solution is summarized as follows. a) We leverage GSE to symbolize only the variables (i.e., registers and memory locations) that are tainted during execution and belong to the set $\mathcal{K}$ as mentioned in 4.2, because the operands of tainted instructions in set $\mathcal{K}$ contain or point to variables that are related to the desired packet fields. b) During monitored execution, we take snapshots of the system states, e.g., when tainting starts (such as when the first tainted byte in the NIC is accessed by the program). This will dump the value of registers and process memory layout. The value of variables that we are not interested in will be passed into GSE to concretize as variables as possible. c) Available packet format information may further limited the range of certain variables, in which case we can apply such conditions to reduce the search space of those symbolic variables.

In particular, as shown in figure 3 (assume that this is a tainted packet), we only symbolize part of the header fields and skip the payload. The fields marked by dark gray will be symbolized while the unshaded fields will keep their concrete values. The hatched area are concrete fields that can help limit the range of symbolized fields. For example, the packet length fields can help set boundaries for the total length of other symbolized fields. Given other specifications of the protocol (such as the min/max/enum value of certain fields), we can further trim the searching space of GSE to accelerate the code exploration.

While dynamic tracing can precisely locate the basic blocks that get executed, and with tainting enabled, it is also able to mark the blocks that are related to specific registers/memory locations of interest, it is easy to see that through one iteration of tracing and tainting, we can only identify the code blocks that process specific network packets and belong to only one execution path of the target feature. There are potentially other execution paths and branches that are also related to the same type of operation but not taken in particular runs. On the other hand, symbolic execution can be employed to explore more paths/branches, providing better code coverage . However, without properly trimming the searching space, it is often faced with a path explosion problem and could easily incur prohibitive overhead in practice. This can be illustrated by reusing the example in figure 1. At node 1, the format of packet is checked and either feature F_a or F_b will be invoked. Without symbolic execution, only one single path ($1 \rightarrow 2 \rightarrow 3 \rightarrow 4$) is considered as the code related to F_a. After customization, the new binary containing only the identified execution path will not be able to process any packet inputs that will lead to node 5. And if symbolic execution is applied without the tainting information, it will start exploration from node 1 and try to symbolically execute all possible paths from there. Significant CPU and memory resources can be required to explore redundant paths such as $1 \rightarrow 6 \rightarrow 7 \rightarrow 4$. Hence, we combine tainting and symbolic execution in CustomPro , by leveraging the tainted variables (such as registers and memory locations) from instruction tracing to guide symbolic execution. We will fix the value in the packet that indicates the feature F_a and symbolize other relevant fields, to explore the code only belonging to feature F_a.

If multiple types of packets that cannot be generated in one run are given as the inputs to feature identification module, CustomPro will perform the above operations one by one and then merge the basic blocks discovered from each iteration. Finally, CustomPro combines the addresses obtained from execution tracing, tainting and GSE to identify a set of instructions that should be kept during binary rewriting. All the basic blocks from 1 to 5 in figure 1 can be discovered by feature identification module, and will be kept in the customized program binary.

5 Feature Rewriting

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

5.1 Instruction Identification

CustomPro will collect traces from different program executions to identify and compute the union of the related feature-constituent functions. This is needed since a single execution trace and symbolic execution may not reach all desired program features (or feature-related code blocks). 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 $\hat{F}$.

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.

5.2 Binary Rewriting

We adopt basic block level binary rewriting in CustomPro . 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 execution to an exit point;
- To prevent the undesired code from getting accessed through malicious operations/attacks such as Return-Oriented Programming (ROP), we replace the undesired basic blocks with "NOP" (except for the shared code and data segments), so they are no longer accessible.

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 1 indicates. If B_2 is the target to be removed, CustomPro will change the call site in B_1 and redirect it to B_3(an exit point) as Arrow 2 indicates. In addition to the instrumentation of control flow, CustomPro will also replace B_2 with "NOP"s to prevent invoking the removed features (and feature-related blocks) at runtime, e.g., through ROP [27].

5.3 Verification

After binary rewriting, standard program fuzzing techniques [42] can be employed by CustomPro 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, CustomPro uses

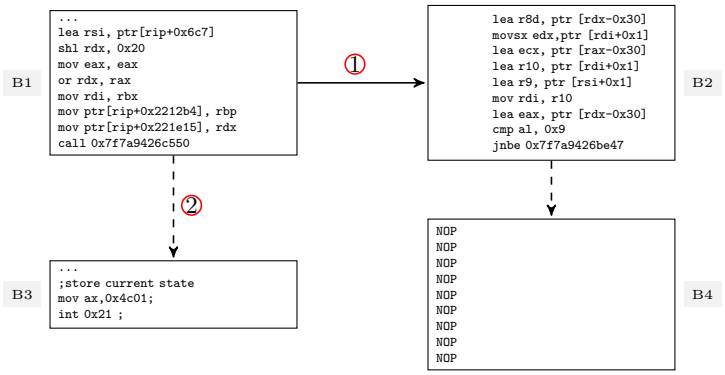

Fig. 4: An illustrative example of implementing feature rewriting on OpenSSL.

$\mathcal{D}$ to confirm the integrity of necessary program functionalities after rewriting, 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 when they are available. If a benign input - that belongs to the desired features in the instrumented binary and should be processed just as previously - 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 be added to the current binary. This step of verification and revision serves as a feedback to the feature identification module to ensure correctness and functionality of the instrumented binary.

6 Implementation

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

Tracing and cross-host tainting: We implement our tracing and cross-host tainting module by modifying TEMU: a) 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 of packets satisfying certain formats or specific fields in the packets. b) We instrument system calls such as $open(), read(), write(), connect(), recv()andsend()$, so that they are able to process and pass taint information across hosts.

Symbolic execution: We use Angr [30] to perform symbolic execution on the static binary. The symbolic execution starts from the memory locations where packets are stored (which can be identified by the tainting information in CustomPro). We also pass the taint tag information to Angr such that the symbolic execution engine will only explore the tainted variables related to target features and avoid the irrelevant features and instructions. We set the end of main function as the ending point of exploration. We also instrument the default exploration mechanism to dump the instructions executed by symbolic engine using the "ins_addr" member in "history.actions".

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 bodies with *NOP*s; (ii) redirects any jumps to the removed basic blocks to a program exit point.

7 Evaluation

In this section, we evaluate the effect of feature customization on two real-world protocol implementations: OpenSSL and Mosquitto. We choose four features from OpenSSL, namely, *heartbeat, client certificate request, renegotiation, encrypt-then-MAC(ETM)*. Since the handshake of SSL protocol is a state machine-based process, and each feature involves the implementation on both client and server, we customize both sides of OpenSSL to eliminate each one of the four target features at one time. Mosquitto implements the MQTT protocol (an IoT messaging protocol) which involves three different entities in a message iteration: broker, publisher and subscriber. After the subscriber sign up for a certain topic, the publisher sends message related to that topic to the subscriber. The broker serves as a middle man and receives all message updates from the publisher then decides which subscribers the messages should be sent to, with respect to the message topics. We choose three features from mosquitto broker and publisher to build different customized versions of the protocol.

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. To evaluate the feature customization on OpenSSL and Mosquitto, we first run the default program to get the number of runtime and static instructions with all target features included. The runtime instructions include all instructions that get executed, excluding dynamic library functions such as glibc code. We further filter out the library code based on the runtime addresses of instructions and mapping information from the */proc/PID/maps*. It is obvious that runtime instructions contain duplicated instructions, as the same basic block in the program binary can be executed multiple times. Hence, we also collect the number of (unique) instructions in the static program binary that are executed during runtime.

7.1 Customizing OpenSSL

OpenSSL is an open-source software that implements SSL/TLS protocols. We use OpenSSL version 1.0.1 for evaluation (the latest version still containing the *Heartbleed* bug). We first collect the number of runtime and static instructions from the default program by running the default *s_server* and *s_client*. Our experiments show that server executes 130212 runtime instructions and 111595 static instructions, while the client side executes 124151 runtime instructions

Table 1: Number of instructions remaining in OpenSSL by removing different features

Removed Feature	# Inst remaining in Server		# Inst remaining in Client	
	Dynamic	Static	Dynamic	Static
HeartBeat	128953	110937	111123	110487
ClientCertificate	121167	103214	107341	95025
Encrypt-then-MAC	128651	110952	122541	110330
Renegotiation	128953	110937	111123	105487

and 111123 static instructions. We remove one of the four target features one-at-a-time to analyze how many instructions are involved in each feature.

HeartBeat: It is one of the well-known bugs in OpenSSL is CVE-2014-0160, namely, *Heartbleed*, which could be exploited by adversaries to steal sensitive data from the SSL server. Heartbleed is rooted in the feature *HeartBeat* in SSL, an extension designed to make sure that each end of the communication is still alive by sending a number of data and expecting the same message being echoed back. However, the receiver of the heartbeat request, rather than checking the actual size of the *HeartBeat* payload, simply allocates a memory buffer with the size declared in the length field of the received packet. If a *HeartBeat* request message is shorter than the claimed value, then extra content will be sent back (up to 64KB) and revealed, containing sensitive memory content from the server. The *HeartBeat* feature can be automatically removed using CustomPro to produce customized program binaries that cannot generate or process *HeartBeat* packets, while other features are still intact. In practice, this provides system administrators with a swift, automatically-generated fix through program customization, without the need or time-overhead to perform full system analyze and patch construction. Our experiments show that 658 static instructions are removed from server and 636 instructions are removed from client.

ETM: Similar to heartbeat, the feature encrypt-then-mac(**ETM**) is also an extension in the process of handshake. After encrypting the plaintext, a MAC(message authentication code) of the ciphertext is calculated and appended to the end of ciphertext. The ETM helps to verify the integrity of the ciphertext. A bug related to ETM has been reported in CVE-2017-3733, where the program will crash if the ETM is defined in the process of renegotiation while it is not in the original handshake (or vice-versa). CustomPro can eliminate such vulnerability by removing the ETM feature from both client (793 instructions removed) and server sides(643 instructions removed) in an automated fashion.

Renegotiation: An interesting case in our study of customizing OpenSSL is about the feature renegotiation. Renegotiation is a feature that enables the connection to change some parameters without establishing a new SSL session so as to save resources. The vulnerabilities about renegotiation reported in CVE database (such as CVE-2009-3555 and CVE-2015-0291) cause the MIIT attack and DoS. However, as CustomPro taint and discover target instructions through the packets/fields, we cannot distinguish the feature renegotiation since the renegotiation request will basically result in another handshake (initialized by a hello

request). As shown in table 1, the customized version of either client or server is the same as the original one.

7.2 Customizing MQTT

Table 2: Number of instructions in Mosquitto after removing unwanted features

# Inst	Removed Features		
	Publisher: insecure	Publisher: publishing file	Broker: loading config
Dynamic	1213	1172	8229
Static	1117	1069	7192

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 in MQTT communications, e.g., broker, publisher and subscriber. The subscriber signs up for a topic via broker, in order to receive messages published by publisher (and through the broker). The MQTT packets contain three fields that include *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 program features are enabled. In this paper, we perform feature customization on Mosquitto ver 1.5.

We run the default Mosquitto broker, subscriber and publisher with all three target features. The numbers of runtime instructions in broker, subscriber and publisher are 8937, 1117 and 1235, respectively. The numbers of static instructions in broker, subscriber and publisher are 7717, 1022 and 1132, respectively. Table 2 shows number of instructions after removing target features.

Among those removed features, some are particularly security concerning. The option "insecure" let subscriber and publisher skip the verification of the server hostname, which means a malicious third party could gain the access to the MQTT communication. The feature "publish file" is a feature that allows the publisher to send files (instead of message updates to subscribers), potentially offering a mechanism for malicious code injection.

8 Discussion

In this section, we discuss some limitations of our CustomPro framework, which will be considered as possible directions for future work.

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 this forward tainting. If the taint source is the outbound packet, then the instructions that generate

such a packet cannot be identified through such forward tainting. To support tainting on outbound packet, backward tainting module can be added to the tainting implementation in TEMU. However, this is only needed when the first packet is generated, because the following packets will all contain the initial taint information (which are provided by our across-host taint propagation), as they are always triggered by some incoming packets from other protocol entities. Hence, one possible future work is to leverage static backward tainting to identify code that generates the first packet, and improve protocol customization.

Rewriting obfuscated binaries: Our current rewriting module performs rewriting on static binary, which requires the precise addresses for instruction rewriting. Obfuscated binaries cannot be supported in our proof of concept implementation. Also, instead of replacing instructions with NOPs and program crash, as a future work, we plan to silent undesired feature invocations and let the program continue execution, such that the (execution of) desired features will not be interrupted by unexpected invocations to undesired features.

9 Related Work

Vulnerability discovery and program customization: As software systems keep expanding and developing new features, more vulnerabilities are introduced to the code. Existing works employ multiple techniques to discover and mitigate bugs in programs such as symbolic execution [32, 23, 40], fuzzing [21], dynamic tainting [35, 9] program customization [15, 13, 5, 6, 17], program analysis [26, 29] and learning-based approaches [36, 39, 40, 19]. 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 [32] 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, achieving a better performance over directed white-box fuzzing and undirected gray-box fuzzing. Hang Zhang et al. 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 [23] also combine tainting ("incomplete" taint propagation) and symbolic execution to improve the runtime tainting performance (by lightweight logging) while still keep necessary information for offline analysis. StatSym [40] 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.

De-bloating: De-bloating has been studied to analyze and mitigate program bloat caused by feature creep [14, 13, 24, 38, 5]. 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 [14]. 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 [13] 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. Most of the previous de-bloating techniques can only work with code in high level languages while CustomPro can be directly applied to binaries.

Binary reuse: Binary reuse has been addressed by several works [44, 33, 45]. The reuse of binary code, different from source code, carries great difficulty. Methods proposed in [3] 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 [18]. 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.

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 [31] and Pin [22] 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 [41] 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 [30] is used to perform symbolic execution together with the tainting information obtained through tracecap.

10 Conclusion

We design and evaluate a binary customization framework, CustomPro , for customizing network protocols. CustomPro aims to generate customized program binaries with *just-enough* features and can satisfy a broad array of customization requirements. Feature identification and feature rewriting are two major modules of CustomPro , for discovering the target features using program tracing and tainting-based symbolic execution, and for modifying the program features through binary instrumentation to obtain a customized program. Our experiment results demonstrate that CustomPro is able to effectively achieve the customization objectives in terms of obtaining an instrumented binary with the necessary functionalities and reducing the corresponding attack surface.

References

1. Tiffany Bao, Johnathon Burket, Maverick Woo, Rafael Turner, and David Brumley. Byteweight: Learning to recognize functions in binary code. USENIX, 2014.
2. David Basin, Cas Cremers, Kunihiko Miyazaki, Sasa Radomirovic, and Dai Watanabe. Improving the security of cryptographic protocol standards. *IEEE Security & Privacy*, 13(3):24–31, 2015.
3. Juan Caballero, Noah M Johnson, Stephen McCamant, and Dawn Song. Binary code extraction and interface identification for security applications. Technical report, CALIFORNIA UNIV BERKELEY DEPT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE, 2009.
4. Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 621–634. ACM, 2009.
5. Yurong Chen, Tian Lan, and Guru Venkataramani. Damgate: Dynamic adaptive multi-feature gating in program binaries. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*, pages 23–29. ACM, 2017.
6. Yurong Chen, Shaowen Sun, Tian Lan, and Guru Venkataramani. Toss: Tailoring online server systems through binary feature customization. In *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation*, pages 1–7. ACM, 2018.
7. Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. Prospex: Protocol specification extraction. In *2009 30th IEEE Symposium on Security and Privacy*, pages 110–125. IEEE, 2009.
8. Weidong Cui, Jayanthkumar Kannan, and Helen J Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *USENIX Security Symposium*, pages 1–14, 2007.
9. Ioannis Doudalis, James Clause, Guru Venkataramani, Milos Prvulovic, and Alessandro Orso. Effective and efficient memory protection using dynamic tainting. *IEEE Transactions on Computers*, 61(1):87–100, 2012.
10. Zakir Durumeric, James Kasten, David Adrian, J Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, et al. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 475–488. ACM, 2014.
11. Zakir Durumeric, James Kasten, David Adrian, J Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, et al. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 475–488. ACM, 2014.
12. Seena Gressin. The equifax data breach: What to do, 2017.
13. Yufei Jiang, Dinghao Wu, and Peng Liu. Jred: Program customization and bloatware mitigation based on static analysis. In *Computer Software and Applications Conference (COMPSAC), 2016 IEEE 40th Annual*, volume 1, pages 12–21. IEEE, 2016.
14. Yufei Jiang, Can Zhang, Dinghao Wu, and Peng Liu. Feature-based software customization: Preliminary analysis, formalization, and methods. In *High Assurance Systems Engineering (HASE), 2016 IEEE 17th International Symposium on*, pages 122–131. IEEE, 2016.
15. Yufei Jiang, Can Zhang, Dinghao Wu, and Peng Liu. Feature-based software customization: Preliminary analysis, formalization, and methods. In *High Assurance*

Systems Engineering (HASE), 2016 IEEE 17th International Symposium on, pages 122–131. IEEE, 2016.
16. Swati Khandelwal. Over 199,500 websites are still vulnerable to heartbleed openssl bug, 2017.
17. Taddeus Kroes, Anil Altinay, Joseph Nash, Yeoul Na, Stijn Volckaert, Herbert Bos, Michael Franz, and Cristiano Giuffrida. Binrec: Attack surface reduction through dynamic binary recovery. In *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation*, pages 8–13. ACM, 2018.
18. Yonghwi Kwon, Weihang Wang, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. 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*, pages 158–169. ACM, 2017.
19. Yongbo Li, Fan Yao, Tian Lan, and Guru Venkataramani. Sarre: semantics-aware rule recommendation and enforcement for event paths on android. *IEEE Transactions on Information Forensics and Security*, 11(12):2748–2762, 2016.
20. Junghee Lim, Thomas Reps, and Ben Liblit. Extracting output formats from executables. In *2006 13th Working Conference on Reverse Engineering*, pages 167–178. IEEE, 2006.
21. Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing. In *Proceedings of the... AAAI Conference on Artificial Intelligence*, 2019.
22. Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.
23. Jiang Ming, Dinghao Wu, Jun Wang, Gaoyao Xiao, and Peng Liu. Straighttaint: Decoupled offline symbolic taint analysis. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pages 308–319. IEEE, 2016.
24. Nick Mitchell and Gary Sevitsky. The causes of bloat, the limits of health. In *ACM SIGPLAN Notices*, volume 42, pages 245–260. ACM, 2007.
25. Gail C Murphy, Albert Lai, Robert J Walker, and Martin P Robillard. Separating features in source code: An exploratory study. In *Proceedings of the 23rd international Conference on Software Engineering*, pages 275–284. IEEE Computer Society, 2001.
26. Jungju Oh, Christopher J Hughes, Guru Venkataramani, and Milos Prvulovic. Lime: A framework for debugging load imbalance in multi-threaded execution. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 201–210. ACM, 2011.
27. Marco Prandini and Marco Ramilli. Return-oriented programming. *IEEE Security & Privacy*, 10(6):84–87, 2012.
28. Anirudh Ramachandran, Yogesh Mundada, Mukarram Bin Tariq, and Nick Feamster. Securing enterprise networks using traffic tainting. *Georgia Inst. Technol., Atlanta, GA, USA, Tech. Rep. GTCS-09-15*, 2009.
29. Jianli Shen, Guru Venkataramani, and Milos Prvulovic. Tradeoffs in fine-grained heap memory protection. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 52–57. ACM, 2006.
30. Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 138–157. IEEE, 2016.

31. Open Source. Dyninst: An application program interface (api) for runtime code generation. *Online, http://www. dyninst. org*, 2016.
32. Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.
33. Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 934–953. IEEE, 2016.
34. Mathy Vanhoef and Frank Piessens. Key reinstallation attacks: Forcing nonce reuse in wpa2. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1313–1328. ACM, 2017.
35. Guru Venkataramani, Ioannis Doudalis, Yan Solihin, and Milos Prvulovic. Memtracker: An accelerator for memory debugging and monitoring. *ACM Transactions on Architecture and Code Optimization (TACO)*, 6(2):5, 2009.
36. Shuai Wang, Pei Wang, and Dinghao Wu. Semantics-aware machine learning for function recognition in binary code. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 388–398. IEEE, 2017.
37. Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. Finding low-utility data structures. *ACM Sigplan Notices*, 45(6):174–186, 2010.
38. Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, and Gary Sevitsky. 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*, pages 421–426. ACM, 2010.
39. Hongfa Xue, Yurong Chen, Fan Yao, Yongbo Li, Tian Lan, and Guru Venkataramani. Simber: Eliminating redundant memory bound checks via statistical inference. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 413–426. Springer, 2017.
40. Fan Yao, Yongbo Li, Yurong Chen, Hongfa Xue, Tian Lan, and Guru Venkataramani. Statsym: vulnerable path discovery through statistics-guided symbolic execution. In *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on*, pages 109–120. IEEE, 2017.
41. H Yin and D Song. Temu: The bitblaze dynamic analysis component.
42. Michal Zalewski. American fuzzy lop, 2007.
43. Angeliki Zavou, Georgios Portokalidis, and Angelos Keromytis. Taint-exchange: a generic system for cross-process and cross-host taint tracking. *Advances in Information and Computer Security*, pages 113–128, 2011.
44. Junyuan Zeng, Yangchun Fu, Kenneth A Miller, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Obfuscation resilient binary code reuse through trace-oriented programming. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 487–498. ACM, 2013.
45. Peng Zhang, Jianhui Li, Alex Skaletsky, and Orna Etzion. Apparatus, system, and method of dynamic binary translation with translation reuse, November 24 2009. US Patent 7,624,384.
46. Qing Zhang, John McCullough, Justin Ma, Nabil Schear, Michael Vrable, Amin Vahdat, Alex C Snoeren, Geoffrey M Voelker, and Stefan Savage. *Neon: system support for derived data management*, volume 45. ACM, 2010.