Principled Unearthing of TCP Side Channel Vulnerabilities

Yue Cao
UC Riverside
Riverside, California
ycao009@cs.ucr.edu

Zhongjie Wang
UC Riverside
Riverside, California
zwang048@ucr.edu

Zhiyun Qian
UC Riverside
Riverside, USA
zhiyunq@cs.ucr.edu

Chengyu Song
UC Riverside
Riverside, USA
csong@cs.ucr.edu

Srikanth V. Krishnamurthy
UC Riverside
Riverside, USA
krish@cs.ucr.edu

Paul Yu
U.S. Army Combat Capabilities
Development Command
Army Research Laboratory
Adelphi, USA
paul.l.yu.civ@mail.mil

ABSTRACT

Recent work has showcased the presence of subtle TCP side channels in modern operating systems, that can be exploited by off-path adversaries to launch pernicious attacks such as hijacking a connection. Unfortunately, most work to date is on the manual discovery of such side-channels, and patching them subsequently. In this work we ask "Can we develop a principled approach that can lead to the automated discovery of such hard-to-find TCP side-channels?" We identify that the crux of why such side-channels exist is the violation of the non-interference property between simultaneous TCP connections i.e., there exist cases wherein a change in state of one connection implicitly leaks some information to a different connection (controlled possibly by an attacker). To find such non-interference property violations, we argue that model-checking is a natural fit. However, because of limitations with regards to its scalability, there exist many challenges in using model checking. Specifically, these challenges relate to (a) making the TCP code base self-contained and amenable to model checking and (b) limiting the search space of model checking and yet achieving reasonable levels of code coverage. We develop a tool that we call SCENT (for Side Channel Excavation Tool) that addresses these challenges in a mostly automated way. At the heart of SCENT is an automated downscaling component that transforms the TCP code base in a consistent way to achieve both a reduction in the state space complexity encountered by the model checker and the number and types of inputs needed for verification. Our extensive evaluations show that SCENT leads to the discovery of 12 new side channel vulnerabilities in the Linux and FreeBSD kernels. In particular, a real world validation with one class of vulnerabilities shows that an off-path attacker is able to infer whether two arbitrary hosts are communicating with each other, within slightly more than 1 minute, on average.

KEYWORDS

TCP; side-channels; model-checking

ACM Reference Format:
Yue Cao, Zhongjie Wang, Zhiyun Qian, Chengyu Song, Srikanth V. Krishnamurthy, and Paul Yu. 2019. Principled Unearthing of TCP Side Channel Vulnerabilities. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, UK*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3319535.3354250

1 INTRODUCTION

TCP side-channels are critical vulnerabilities that can be exploited by adversaries towards launching dangerous attacks. Prior studies have demonstrated that TCP side-channels can be exploited by off-path attackers to perform idle port scans [16], to estimate the round trip time (RTT) of a connection [1], or to infer how many packets were exchanged over a connection [11]. They even allow attackers to hijack connections between a client and a server [7, 11, 18, 37, 38], These side-channels are an artifact of unforeseen code interactions, can arise with the deployment of large code bases, and are subtle and hard to find.

Most of the aforementioned side-channel vulnerabilities are discovered manually by domain experts. While manual analysis has been immensely useful in discovering and patching such subtle vulnerabilities, it requires a significant effort, and is thus not scalable and cannot guarantee the elimination of such vulnerabilities. In this work our goal is to *develop a principled approach to automate the discovery of such hard-to-find TCP side-channel vulnerabilities*.

In principle, TCP side-channel vulnerabilities are violations of the *non-interference* property [21] between simultaneous TCP connections, i.e., the existence of one connection can have an observable effect on the other connection(s). Thus, off-path attackers can use their own connections to the server to infer the properties (e.g., sequence number) of a targeted TCP connection between a victim client and the same server. Specifically, an attacker can send spoofed packets with guessed properties to the server. If the guess is correct or close, the spoofed packet will cause a change in the state at the server which in turn, causes changes pertaining to the attacker's own connection to the server.

Based on this observation, we design a tool SCENT, to find TCP side-channels in a complex code base with very little manual intervention. At a high level, SCENT detects TCP side-channel vulnerabilities by detecting violations of the non-interference property between connections. In particular, it uses two instances of the same server (TCP stack), where the only differences are in the security sensitive properties (e.g., sequence number, acknowledgement number, or port) of an idle (victim) connection. It then sends a set of packets (inputs) to the two servers. If the responses from the servers are different, then SCENT has detected a violation of the non-interference property.

While this approach is intuitive, the challenging part is determining what kind of packets to send in order to induce such a violation. Given the large search space of possible combinations of TCP packets, popular dynamic testing techniques like symbolic execution and fuzzing all face efficiency problems. In this work, we resort to bounded model checking [14, 29, 31] to drive an analysis to answer this question. Compared to bounded testing [32, 42] (i.e., blindly enumerating all possible packets up to the bound), bounded model checking enjoys the benefit of state deduplication and is thus, much more efficient (see §8 for more details).

Unfortunately, applying model checking to a real-world TCP stack implementation is non-trivial. First, we need to prepare a self-contained model that is amenable for model checking (otherwise the code base is simply too large). Previously, the work by Enasfi et al., [16] has adopted model checking to detect non-interference property violations in the network stack. However, due to the complexity of implementation level code, they had to manually craft a much simplified abstract model for the analysis. Such an approach, while useful in their context of interest (discovering idle port scan techniques), cannot guarantee that subtle TCP side-channels buried in complex implementations like the Linux kernel, will not be (unintentionally) removed during the abstraction. To avoid this problem (high false negatives), we opt to use the unmodified TCP stack implementation for analysis and only abstract away code that is outside the core TCP stack.

The second challenge is state explosion. TCP implementations from real-world kernels contain many variables; if we blindly mark all the variables as *states*, then any change to any variable will be deemed as a new state. However, if a variable is never "shared" between two connections, it cannot leak any information and is thus, not interesting to track. To solve this large state space challenge, we develop a conservative static analysis within SCENT to safely reduce the state space.

The last challenge is that bounded model checking has bounded code/state coverage and hence, cannot detect all vulnerabilities. For instance, the TCP side-channel discovered by Cao et al., [7] requires sending 100 packets, which is way beyond the capability of bounded model checking. To solve this problem, we developed a program transformation technique to automatically simplify the model as a way to improve the code coverage. In particular, we observe that many uncovered cases relate to branches that compare an attacker-controllable value with a fixed value (e.g., the global rate limit exploited in [7]), and the problem is that the bounded input space cannot drive the variable side of the branch to go beyond the fixed threshold. Based on this observation, SCENT automatically identifies such branches and downscales the fixed threshold so that both branches can be visited during a subsequent iteration of bounded model checking.

To demonstrate the effectiveness of our approach, we have implemented a prototype of SCENT and created two realistic TCP models, one based on the Linux kernel (version 4.8.0) and the other one based on the FreeBSD kernel (version 13.0)[1] We applied SCENT on these two models and found 12 new side-channel vulnerabilities. A real world evaluation shows that in particular, with one of the classes of vulnerabilities discovered, an off-path attacker is able to infer whether two arbitrary hosts are communicating with each other, within slightly more than 1 minute on average. The evaluation results also show that our transformation step is critical for finding these side-channels—none of them can be found without the transformation. Besides, we also did not observe any false positives during our evaluation.

Contributions. Our contributions can be summarized as follows:

- We design and implement SCENT, a system that finds subtle TCP side-channels by detecting violations of the non-interference property between TCP connections, using model checking as a basis.
- We developed several techniques to automate the process of creating self-contained code amenable for use with an off-the-shelf model checker, from real-world kernels that keep the core TCP implementation intact. We applied these techniques to the Linux and the FreeBSD operating systems and open sourced the extracted models at [41].
- We developed a code-transformation-based model simplification technique that improves code coverage for bounded model checking.
- We applied SCENT to the Linux and the FreeBSD TCP models and found 12 new side-channel vulnerabilities. We open sourced our system and released the complete details of findings at [41].

2 BACKGROUND

In this section, we briefly describe the non-interference property and why it is relevant to the problem of interest. Subsequently, since we use model checking as a basic building block, we provide relevant background in brief.

The non-interference property. In recent decades, the non-interference property [21] has been widely used as a requirement to prove that neither explicit nor implicit information leakage can occur in a scenario of interest. Because side-channels are a consequence of information leakage, the non-interference property can be used as a verification condition to ensure that they do not exist. If the property is violated, it indicates the potential presence of an information leak, which can in turn lead to an exploitable side-channel vulnerability. With regards to the context of interest, if this property holds, it implies that a state change on a given connection does not (implicitly or explicitly) become observable in another connection.

Ensafi et al. [16] applied model checking to verify the non-interference property in the TCP/IP stack, towards finding side

[1] SCENT can be applied to any OS kernel as long as the source code is available. Therefore, SCENT can be potentially applied on Windows in its internal environments.

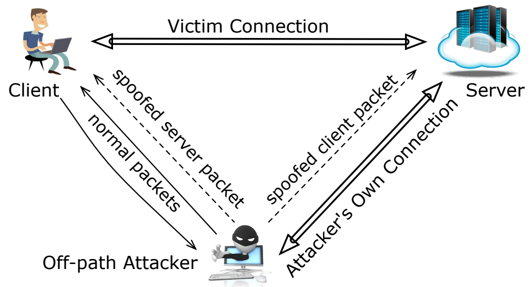

Figure 1: Threat model

channel vulnerabilities relating to idle port scans. While they find two port scan vulnerabilities, we point out that they use model checking more like a validation tool instead of a tool to discover these; they heuristically specify the scope of the TCP code (to only consider the specific shared resources across connections) and then manually build the model.

In this work, we seek to perform non-interference analysis in more general attack scenarios. Importantly, since a manually abstracted model like that in [16] is very approximate and may miss vulnerabilities that exist in real code, we explore applying model checking to real TCP implementations from commodity kernels.

Software model checking. Model checking [13, 40] exhaustively checks if a given model of a system satisfies a given formal property. If violations are encountered, the model checker outputs counter examples which enable the locatation of where a violation has occurred with relative ease. Model checking methods can be broadly classified into two categories viz., those that use abstraction (e.g., SLAM [3], BLAST [22], Event-Driven Software Verification [25]) and those that are applied directly on implementations (e.g., VeriSoft [20], CMC [34] [33], and Model-Driven Software Verification [24]). Since the former relies on extracting an abstraction from the real code (and thus can result in significant approximation), we use the second category to verify the non-interference property in our work.

Specifically, the basis for our work is a TCP event-driven execution model that we build. Different from previous work relating to the use of model checking with an abstracted state machine of either TCP or the network stack (e.g., [33, 34]), we check for possible violations of the non-interference property in real TCP implementations. Our model is much more complicated since we have to look at verifying a property relating to connection interactions (as discussed later our model contains 4 live TCP connections with 6 different sockets). Furthermore, we need to address challenges relating to making the model self-contained (to ensure that it can be used with an off-the-shelf model checker) and concise (without which the complexity of the code will make it untenable to the model checker). Unfortunately, even just the core TCP stack implementation is too complex for the model checker to exhaustively check all possible states of the code. For this reason, we can only perform bounded model checking and therefore the conclusions (existence or absence of violations) are only applicable to a bounded set of states instead of the entire code base.

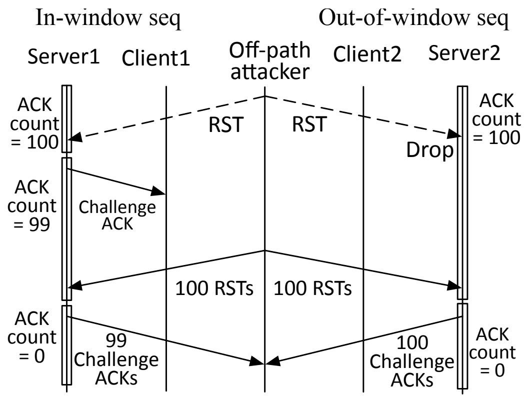

Figure 2: An illustrative TCP Side-Channel Vulnerability.

3 THREAT MODEL

Our threat model is that of an off-path TCP attacker as shown in Figure 1. We consider 3 hosts viz., a victim client, a victim server, and an off-path attacker. The attacker can either send packets on its own connection to the server, *or* send spoofed packets with the victim client's IP address or a victim server's IP address. Different from a Man-in-the-Middle (MITM) attack, the off-path attacker can neither eavesdrop nor inject packets into the victim connection. Instead, it attempts to exploit a side-channel vulnerability to infer the state of the victim connection based on the packets sent/received on its own connection. Specifically, it could target the inference of (a) the *port number* of the victim client (the server's port number is usually known), (b) the *sequence (SEQ) number* from the client, and/or (c) the *acknowledgement (ACK) number* expected by the server. By inferring just the port number, the attacker can determine if there is an established victim connection between the server and the client. With the port number and the SEQ number expected by the server inferred, the attacker can launch a DoS attack by sending a packet with the reset (RST) flag (and correct SEQ number) to terminate the victim's connection. If all the three attributes are inferred, the attacker can hijack the victim connection and inject malicious payloads as shown in [7]. Note that any machine around the world can launch an off-path attack, as long as it is able to send spoofed packets with the victim client's (or server's) IP address.

Previous TCP inference attacks [7, 11, 37, 38] follow a "guess-then-check" strategy. Specifically, during the guess phase, a spoofed packet is sent with a guessed value (for either or a combination of the port number, SEQ number and/or ACK number). A correct guess will be "accepted" by the TCP state machine thus causing it to transit into a state that is different from that due to wrong guesses. During the subsequent check phase, the attacker exploits the side-channel vulnerability to leak the state transition of the victim's connection, which allows the attacker to tell whether the guess is correct or not. Like in these efforts, the focus of this work is on identifying similar "software-induced"[2] side-channels but by using a more principled approach.

An illustrative TCP side-channel vulnerability. To illustrate how an off-path attacker can exploit a side channel vulnerability to determine the state of a victim connection (in terms of port number,

[2]Other types of side channels, such as timing based ones [11], are out of the scope.

SEQ number or ACK number) consider the recent example from [7]. Figure 2 captures this example wherein the off-path attacker infers the expected SEQ number of the victim connection to the server.

To understand how the attack works, consider two cases. In the first case, the SEQ number guessed by the attacker is within the "receive window" (in-window) of the server while in the second case, the SEQ number is out-of-window. The attacker sends a spoofed RST packet with a guessed SEQ number. If the number is in-window, the server responds to the victim with a "Challenge ACK" packet to ask the client to confirm the RST. Since the victim client did not really send the RST packet, it will simply discard the Challenge ACK packet. To control how many Challenge ACKs can be sent within a time period, the Linux kernel maintains a *global shared counter* (equal to 100 prior to the work in [7]). Thus, when the attacker subsequently sends in-window RST packets on its own connection (one after the other as shown in the bottom part of the figure), it gets back 99 Challenge ACKs; in contrast, if the spoofed RST packet is out-of-window, the attacker will receive 100 Challenge ACKs. This difference/side-channel can then be used to infer whether the guess is correct or not.

What is evident in the above example is that, by observing the number of Challenge ACK responses from the server on its own connection, the attacker can distinguish between two cases with regards to its spoofed packet viz., whether the SEQ number guessed is within the server's receive window or not. Thus, this is a violation of the non-interference property i.e., the state of the client's connection influences how many Challenge ACKs are received by the off-path attacker.

4 SCENT OVERVIEW

In this section, we provide an overview of our system SCENT and its core innovation.

4.1 Workflow

Figure 3 shows the overall workflow of SCENT. Specifically,

- Taking the source code of a commodity OS kernel as input, the Model Generator (§5) generates a self-contained model[3] amenable for application of an off-the-shelf model checker and pushes this initial model into a queue.
- The Non-interference Checker (§6), at each step, takes one self-contained TCP model from the queue, constructs an attack scenario, and executes bounded model checking to verify the non-interference property between connections.
- If violations are found by the model checker, validated counter-examples are output as the proof-of-concepts for possible TCP side-channel vulnerabilities inside the kernel's TCP stack implementation.
- Finally, to mitigate the limited code coverage of bounded model checking, the Model Transformer (§7) automatically generates a new, downscaled model and pushes it into the queue for the next round of analysis.

[3]Note that we only abstract code irrelavent to TCP stack; previous work abstracts the TCP stack itself.

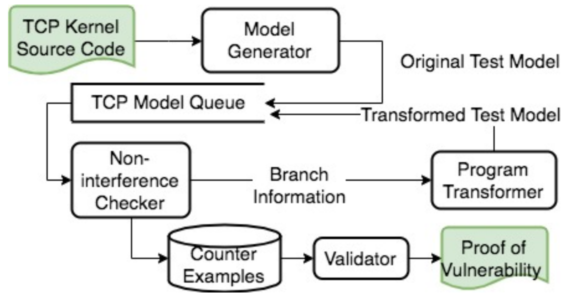

Figure 3: Overview of SCENT's workflow.

4.2 Automated downscaling

While applying bounded model checking to TCP implementation as a way to find non-interference violations is not entirely new, SCENT solves an important and non-trivial problem. In principle one will need to send an extremely large sequence ($\approx \infty$) of packets in order to excavate all possible violations of the non-interference property. Unfortunately, we point out that due to the complexity of commodity kernels' TCP implementations, even a relatively small sequence of TCP packets can lead to an explosion of the state space that cannot be explored by the model checker with limited computation resources (CPU time and/or memory). As a result, the bound we can afford is considerably small (e.g., only 3 packets in our evaluations); otherwise, the model checker will either exhaust memory or take a prohibitively long time to finish. This further translates to limited code coverage and impacts the effectiveness of SCENT (i.e., it cannot detect side-channel vulnerabilities in uncovered code). For example, the vulnerability illustrated in Figure 2 cannot be detected as triggering it requires sending 100 RST packets. In fact, side-channels are more likely than not, triggered by such uncommon sequences of packets. SCENT copes with this scalability issue via a novel technique we call *automated downscaling*.

Our observation is that the TCP code base contains many checks (branches) that compare attacker-controlled variables against either some constant values or variables that remain the same during model checking. Due to the limited input bound, those attacker-controlled variables have limited value ranges. When the attacker-controlled value range does not overlap with the fixed value (cover both sides), only one branch can be covered. However, such linear relationships between an attacker value and a fixed value can be satisfied easily by downscaling the fixed value (i.e., moving it torwards the attacker-controlled value range). More importantly, this transformation will not change the fundamental behavior of the TCP implementation: without downscaling, the relationship can still be satisfied, but simply takes significantly longer inputs and therefore times.

To further elucidate this observation, let us revisit the example from Figure 2. The side-channel relies on the global Challenge ACK rate limit (a variable with fixed value 100) and the attacker has to send 100 packets in total (one spoofed and 99 on its own connection in the example), to trigger the information leakage. To find this vulnerability, intuitively, the model checker will have to examine what happens when the TCP code base has received different numbers of packets which have the RST flag set and are in-window (it has to perform 100 such checks). Unfortunately, this is not possible during our bounded model checking because we can only increase the counter from 0 to 3. However, if we were to simply (artificially) change this rate limit to say 2, then we will be able to trigger this vulnerability and observe the difference.

Furthermore, the advantage of this approach is that it also inherently reduces the required input space we need to enumerate. For example, one can reduce the space of possible SEQ numbers (from 2^{32} to a much smaller value) by downscaling other fixed constants (e.g., the receive window size). This also contributes to a drastic reduction in the time-complexity associated with our analysis.

Practical Realization. To practically realize automated downscaling, we pursue an iterative approach (alluded to in the workflow described in § 4.1). This approach is driven by the key insight that there is a tight coupling between the input space (i.e., length of our input packet sequence and the space of the fields in the TCP header such as SEQ number space) and the values to which the limits in the code are to be changed. In the example above, changing the limit to 2 requires the attacker to send a sequence of two packets. If on the other hand, we knew that the attacker had a packet sequence of length 5, the limit could be anywhere from 2 to 5.

Given this, for ease of realization, to begin with, we fix the length of the input packet sequence and the sizes of the header fields in each, but do not modify the TCP code that is input to the model checker. During the model checking phase, we log information relating to what parts of the code (what branches) are not covered because of control statements relating to such limits. We then use concolic execution to establish transformations of such constraints (guided by the constraints imposed on the input packet sequence) that may make such coverage viable (using the program transformer module shown in Fig. 3). The transformed model is then considered for bounded model checking. We iterate the process until we either (a) do not find any additional transformations that we can perform or (b) we exceed a pre-specified time limit.

5 MODEL GENERATOR

In this section, we describe in detail how we address the challenges in constructing a standalone TCP code base that can be input to the model checker and how we initialize variables to ensure that the model begins with a valid and consistent TCP state.

In principle, one can apply the design principles from [24] to construct a test model, which combines a test-harness and the real kernel code with an initial state. Given this initial state, the test harness would enumerate a sequence of packets as input, and calls the TCP packet reception code to explore the set of reachable states. Here, the state of the model is defined as the union of internal states at a host, and is determined by the values of global variables and heap objects that are reachable by the connection object (i.e., the socket). Unfortunately, applying model checking directly on a kernel code in its entirety, is not practical. This is because model checking has high associated time complexity, and using the entire kernel code base as the model can make the analysis prohibitively costly. More importantly, many of the paths explored by the model checker will have no bearing on what we seek to analyze. Last but not least, there is significant non-determinism in real TCP implementations which will interfere with the model checking.

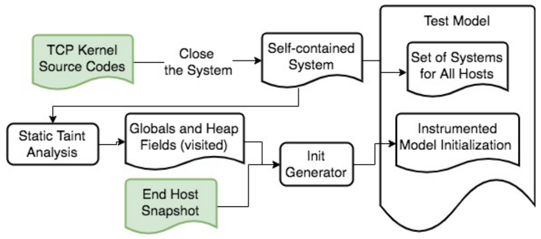

Figure 4: Workflow of the Model Generator.

Therefore, assembling a standalone TCP implementation without any kernel dependencies becomes important for the feasibility of our approach. However, extracting the TCP code from a kernel is challenging given the fact that the TCP code interacts with the rest of the kernel via complex interfaces. We solve this challenge by identifying boundaries where the code can be pruned and manually constructing stub implementations to close the boundaries.

In addition to generate a self-contained TCP code base for the model checker, another challenge is how to properly initialize the model. SCENT solves this challenge by automatically extracting correct values from a memory snapshot.

5.1 Building a Standalone TCP Model

The high-level guideline for building a standalone TCP code base is that we want to make sure that all the code related to the TCP stack remains exactly the same as in the target kernel, while code not related to the TCP stack should be minimized/abstracted. Following this guideline, we use a simple worklist-based, semi-automated approach to gradually grow the code base until the whole TCP stack is included.

(1) We initialize the worklist with the entry function of the TCP layer when a packet is received (e.g., `tcp_v4_rcv` in Linux and `tcp_input` in FreeBSD).
(2) We try to remove one function from the worklist. If the worklist is empty, we terminate the process; otherwise we move on to the next step.
(3) We check if the current function belongs to the TCP layer (based on our domain knowledge). If so, we copy the whole function to the standalone mode and move on to the next step; otherwise we manually write a stub function to abstract it and go back to Step (2).
(4) We find all the callees of the current function and add them into the worklist and go back to Step (2). For indirect calls, we manually resolve the target based on domain knowledge.

Note that because in our attack scenario (§6.2) we keep the victim connection idling, our current model excludes functions relating to sending packets on that connection.

5.2 Initializing the Standalone TCP Model

Because our TCP model is built using partial kernel code starting at an entry function, we need to initialize what we call *environmental variables* at this entry point. This is to ensure that the initial state provided to the model checker is correct and consistent with TCP

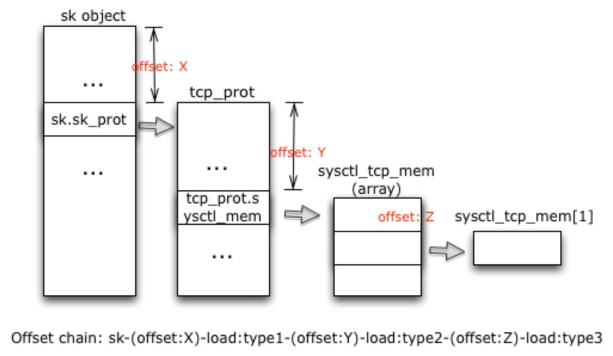

Figure 5: Using offset chains to locate the target variables during initialization.

executions. Such environmental variables include the entry function's arguments, global variables, and heap objects that may be accessed or reachable by the code extracted above. We point out that there is no need to initialize local variables or heap objects that are allocated (and initialized) during the execution of the model.

Manual identification of all the variables that have to be initialized is not only an onerous task but is also error-prone. Thus, we develop an automated procedure to initialize them based on a memory snapshot from a running kernel, which is captured when the entry function is invoked, Because our standalone model runs in user-space, values from the snapshot cannot be directly used as they could be pointers. So our method needs to (1) identify all accessible variables and their types, (2) locate each target variable in the snapshot (i.e., determine its address), (3) extract its value according to its type and size. Finally, this will allow us recreate the variables and initialize their values in the model checker.

We achieve these goals via a process that is similar to previous work on recovery of kernel objects from memory snapshots [8]. First, starting from anchor variables (i.e., entry function's arguments and global variables explicitly referred to in the model), we use static taint analysis to recursively identify all accessible/reachable heap and global objects by following pointers. Due to the existence of typecasting, we identify pointers in two ways: (1) based on the variable/field's declaration type and (2) based on the use of the variable/field.

To locate variables inside the memory snapshot, we maintain the point-to relationship between kernel objects in a data structure that we call *offset chain*, which tracks how each variable is derived from an anchor variable and the used type associated with the variable. The offset chain allows us to traverse the snapshot and recover the corresponding variables.

Once we locate a variable inside the snapshot, we extract its "initialization" value based on whether it is a pointer or not. For non-pointer variables, we will directly use its value from the snapshot; for pointer variables, we will allocate the target variable statically in the model checker and assign the target object's address as the initialization value. One particular challenge in this step is how to decide the size of the variable if its type is an array with unknown size. For example, in Linux, the packet header pointer `skb->head` is a pointer to an unsigned char, which can be used to visit the packet payload with a specific offset (via a value of header field `doff`). As associating size with a pointer is a hard program analysis problem, currently we solve this challenge manually.

Figure 5 illustrates this process via an example. In the figure, each offset depicted represents the address offset between the beginning of a heap object and the current field in the object. Given the base address in the snapshot and `offset:X`, our method can obtain the corresponding field value. If the field is a pointer, its value can be further dereferenced in the snapshot to locate the next (new) heap object. Given this new heap object's base address and `offset:Y`, a new field can be located and so forth.

6 NON-INTERFERENCE CHECKER

In this section, we describe how we detect violations of the non-interference property between two TCP connections.

6.1 Constructing the attack scenario

To detect violations of the non-interference property between two connections, we craft an attack scenario that is similar to what was captured in the illustrative example (Figure 2). The scenario consists of two servers (Server1 and Sever2), two clients (Client1 and Client2), and an attacker (Figure 6). Both servers and clients use the same self-contained model from the Model Generator. A connection between Server1 and Client1, and Server2 and Client2 is initialized before testing. The two connections are identical except *a specific secret* relating to the victim connection. We use the connection between Client1 and Server1 to model the case when the guessed secret is correct, and use the connection between Client2 and Server2 to model the case when the guessed secret is wrong.

Ideally, to find all possible side-channel vulnerabilities, the attacker (test-harness) should exhaustively generate all possible input packet sequences, including both spoofed packets (with the IP address of the victim clients) and packets on its own legitimate connection. Unfortunately, given the unbounded search space, this is simply infeasible. So our test-harness only enumerates all possible input packet sequences up to a bound (i.e., performs bounded model checking). Once the test-harness generates a packet sequence, it sends the same sequence to both servers. Because only the secret attribute is different for the two victim connections, if the packets received from the two servers are different (including the number of packets, the pattern/order of received packets, the contents, etc.), the non-interference property is violated and the secret is leaked. The counter-example (packets being sent from the attacker) is then reported as violations by the model checker.

To reduce the effort of the attacker and to make the model more deterministic, we keep the victim connections "idle" during the model checking (i.e., neither the victim client or the server will actively send packets in our model). By doing so, we can be sure that differences in the received packet sequence are indeed caused by the spoofed packet sequence. If the server and client are actively exchanging packets, it becomes hard to identify a violation (differences may simply be due to those exchanges).

6.2 Secrets of interest

Our focus in this work is on identifying side-channel vulnerabilities that result in the leakage of three specific secret attributes of a

[4]We set the secret as whether the specific port is being used by the victim connection.

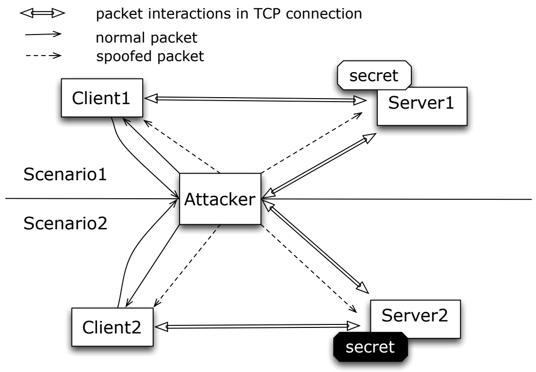

Figure 6: Our setup for the scenario relating to non-interference property verification.

Table 1: The 6 different secret settings of interest (The initial state captures the victim socket state at the server side)

Initial State	SYN-RECEIVED			ESTABLISHED		
secret	port no.[4]	SEQ	ACK	port no.	SEQ	ACK

connection viz., *port number*, *SEQ number*, and *ACK number*. While there could be other sensitive information (e.g., that reveals the socket state), we focus on these since they have been shown to be exploitable for DoS or connection hijacking [7]; however, our approach can be used to infer the leakage of other secrets.

To formalize, we require our model checker to verify the following three properties with respect to non-interference. The sequence of packets received by the attacker is identical with respect to the two servers, regardless of :

- The port numbers used in the victim connection;
- The SEQ numbers used in the victim connection; and,
- The ACK numbers appearing in the victim connection.

While previous work only examines if and how these secrets are leaked when the victim connection is in the ESTABLISHED state of TCP, we extend the scope to include cases wherein the victim is in the SYN-RECEIVED state (i.e., during the three way handshake). This is because in this state, if the attacker can acquire the information of interest, it can infer whether the client tried to establish a connection with the server, or even potentially establish a fake connection itself (by sending a spoofed SYN packet with the client's IP address – note that in this case, the SYN-ACK returned by the server to the victim client will be likely filtered by NAT and stateful firewalls). The latter attack can be serious in practice, since the attacker if successful, can subsequently inject malicious data on to the server pretending that the data came from the spoofed client's IP address. Thus we have a total of 6 secrets (in the two states combined) listed in Table 1.

6.3 Bounding the input packet sequence

In this work, we focus on the control bits and the secret of interest (port, SEQ, or ACK number) in the TCP header. All other fields are fixed, and are copied from snapshots from real connections. We exclude the TCP header options in this work since not all systems support these. Table 2 captures the bounded input space that the attacker (test harness) in our scenario, generates. For fields that have small value ranges, such as the TCP flags, we enumerate all the possibilities, (except for FIN and congestion related ones). For fields that have larger value ranges, we determine the range as described below.

Recall that in our attacker scenario (Figure 6), we use the connection between Client1 and Server1 to model the case when the guessed secret is correct (the guess will automatically be wrong for the connection between Client2 and Server2 because the secrets are different). Therefore, for the field related to a secret of interest, the value range is decided based on the concrete values from the connection between Client1 and Server1. Assume that on Client1's side, the port number, SEQ number, and ACK number are P, S, and A, respectively. We will always set the port number of all generated packets to P, because this allows us to exercise the scenario where we made a correct guess of the port number of the connection between Client1 and Server1, and a wrong guess of the port number of the connection between Client2 and Server2. For the SEQ number and ACK number, because the TCP stack performs a range check, we want to simulate cases where the guessed number is close to, but not equal to the correct number. For this reason, we enumerate the range $[S-2, S+2]$ and $[A-2, A+2]$. Currently, we limit these variables to this range, because with our automated downscaling (of the receive window), the enumerated cases are enough to explore all three of the following cases with respect to those numbers: (a) outside the receive window, (b) exact match, and (c) within the receive window.

We currently limit the range of the payload size to $[0, 2]$. This range allows our model transformer to downscale packet size related checks; yet, it will not significantly increase the input space or the state space.

Finally, we determine the packet length through empirical experiments, i.e., given that the ranges of each packet's fields are fixed, we try to enumerate as many packets as possible until the model checker either exhausts the memory or takes a prohibitively long time to finish. On our evaluation platform, we can only enumerate a maximum packet sequence length of 3.

6.4 Deduplication

Since different input sequences can trigger the same vulnerability, the counter-examples can also be duplicated. To find distinct vulnerabilities, we follow a similar approach as semantic crash bucketing [43]: given a counter-example, we use the branch trace recorded for the model transformer (§7) to locate the key branch/constraint that leads to the different behavior and "patch" the branch so that the counter-example will no longer yield the different behavior. One can consider this as the opposite process compared to our model transformation process. Then, we run all the counter-examples again. All the other counter-examples that no longer yield the different behavior will be considered to be duplicates.

7 MODEL TRANSFORMER

In this section, we describe how we practically realize the vision of automated downscaling using an iterative approach. We begin

Table 2: Packet fields enumeration ranges. C1 means the corresponding value used by Client1 in our attack scenario. Packet with IP equal to C1 is spoofed packet, while packet with IP equal to Attk is on attacker's own connection.

	Packet Fields									Length of
	IP	SEQ Num	ACK Num	SYN	ACK	RST	PSH	URG	Payload Size	Packet Sequence
Original range	[0, 2^32)	[0, 2^32)	[0, 2^32)	0/1	0/1	0/1	0/1	0/1	[0-1460)	Infinite
Bounded range	C1/Attk	[C1-2, C1+2]	[C1-2, C1+2]	0/1	0/1	0/1	0/1	0/1	[0-2]	3

with a limited input space and execute bounded model checking. During the process, we log information relating to the code that are not covered but relate to fixed limits (discussed earlier). We then apply concolic executions to determine how these limits must be transformed (flipped), to make the code coverage feasible while adhering to the inherent constraints imposed by the chosen, limited input space. The transformed model is then re-considered (in the next iteration) and the process is repeated until we do not find any new transformations that can be performed or when we exceed a pre-specified time limit. We describe these steps in detail below.

7.1 Identifying target branches

Given the above premise, our first challenge is to locate branches we aim to flip. To do so, we instrument our model so as to trace all the branch instructions and dump their conditions during the model checking phase. Then we parse the trace and look for branch conditions (a relation operation like $<, >$) that have one operand that varies (e.g., a stateful variable), while the other operand is always a fixed value. Next, we check if both the true branch and false branch are visited during model checking; if only one branch is visited, we have found a target branch.

7.2 Determining expected values

After identifying the target branch, the next step is to determine the expected value. Given that the other operand imposes a range $[l, h]$, we have two general options: we can either move the fixed value to the other side of the range ($> h$ or $< l$) or move it to the middle of the range. In this work, we choose to move the value to the middle of the range for the following reason. The goal of the non-interference checker is to find a behavior that differs between Server1 and Server2 when handling the same input sequence. One reason such a difference can arise is Server1 and Server2 taking different paths at the same branch. For a target branch, the input sequences we enumerate can only go down one path, with both servers; otherwise we would have observed these cases. So if we move the fixed operand to the other side of the range, the input sequence still can only go down only one path, which is not particularly useful for revealing potential different outputs.

7.3 Identifying targets for transformation

The goal of our model transformer is to rewrite the program so that our limited input packet sequence can visit *both* the true and false conditions of a branch. After identifying a target branch, there are multiple ways to rewrite the program to achieve this goal. One way is to replace the relational operation with one that compares the variable operand with a smaller but *fixed constant*. However, this simple approach could introduce inconsistencies when the operand with the fixed value is derived from one or more program variables that are also used in other constraints (e.g., branch conditions). To avoid potential false positives or false negatives introduced by such inconsistencies, we choose to modify the source variables during initialization, instead of directly patching the branch.

There are two general approaches to identify the source variable(s), data-flow (taint) analysis and symbolic execution. Because the source variable(s) could go through a series of operations before being used in the target branch (e.g., `limit+10`), we choose symbolic execution. This approach provides us with a symbolic formula expressing the relationship between the source variable(s) and the value used in the target branch. Therefore, given an expected value to be used in the target branch, we can consult a SMT solver to automatically determine the corresponding value(s) to which the source variable(s) need to be set during initialization. Moreover, symbolic execution also allows us to collect all path constraints prior to reaching the target branch. By adding these constraints while querying for suitable initialization value(s), we can ensure that the new value(s) will not break path constraints leading to the target branch.

In brief, SCENT uses concolic execution to determine (a) which variable(s) should be modified during initialization and (b) what is the value(s) to which it must be initialized. Because we perform concolic execution over a single trace (only to collect the symbolic formula relating to a branch predicate), we point out that we do not have a problem of path explosion. A sketch of the process is as follows:

(1) SCENT conservatively symbolizes all variables that are related to the system's internal states. (i.e., all the global and heap objects found in §5).
(2) SCENT applies concolic execution wherein one path recorded during model checking is followed to reach a target branch constraint.
(3) SCENT checks if the operand with a fixed value is symbolic (i.e., derived from internal states). If not, we directly patch the constant and exit; otherwise we move on to the next step.
(4) SCENT queries a SMT solver for a feasible assignment to the internal states such that (a) the path constraints are satisfied and (b) the operand used in the branch will fall into the range of the variable operand.
(5) If the solver can return an assignment, SCENT modifies the model initialization procedure to assign the values returned from the SMT solver to the related internal states; otherwise it tries to find another recorded path that can lead to the target branch and goes back to Step (2).

Table 3: Side-channel vulnerabilities discovered by SCENT with different initial secret settings.

OS Kernel	Index-ClassID	Key Constraint that Causes Violations	Different Outputs	6 Secret Settings in Table 1						Transfor -mation Required	New
				SYN_Recv			Established				
				port	SEQ	ACK	port	SEQ	ACK		
FreeBSD 13.0	1-A	V_icmp_rates[3].cr.cr_rate <V_icmplim	RST pkt vs NULL	+			+			Y	Y
	2-A		RST pkt vs NULL	#	#	#	#			Y	Y
	3-A		RST pkt vs NULL	+			+			Y	Y
	4-A	V_icmp_rates[4].cr.cr_rate <V_icmplim	RST pkt vs NULL	+			+			Y	Y
	5-B	sch->sch_length >= V_tcp_syncache.bucket_limit	NULL vs RST pkt				+			Y	Y
Linux 4.8.0	6-C	tcp_memory_allocated <sysctl_tcp_mem[2]	ACK pkt vs NULL	#	#	#	#	#	#	Y	Y
	7-C	tcp_memory_allocated <sysctl_tcp_mem[1]	Immediate ACK vs Delayed ACK	#	#	#	#	#	#	Y	Y
	8-C		ACK pkt with different window size	#	#	#	#	#	#	Y	Y
	9-B	inet_csk_reqsk_queue_len(sk) >= sk->sk_max_ack_backlog	SYN-ACK pkt vs NULL	+			+			Y	Y
	10-B	inet_csk_reqsk_queue_len(sk) >= sk->sk_max_ack_backlog	SYN-ACK pkt vs NULL		#	#				Y	Y
	11-B	sk->sk_ack_backlog >sk->sk_max_ack_backlog	SYN-ACK pkt vs NULL	#	#	#	#			Y	Y
	12-D	challenge_count <sysctl_tcp_challenge_ack_limit	ACK pkt vs NULL	#	#	#	#			Y	Y
	13-D		ACK pkt vs NULL	+			+			Y	N
	14-D		ACK pkt vs NULL	*	*		*	*		Y	N
	15-D		ACK pkt vs NULL	#	#	#	#	#	#	Y	N

+: correct port number required to trigger the violation
*: correct port number and SEQ number (in-window) required to trigger the violation
#: correct port number, SEQ number (in-window) and correct ACK number required to trigger the violation

8 EVALUATIONS

In this section, we evalute SCENT on two OS implementations, viz., Linux 4.8.0 and FreeBSD 13.0, with the goal of answering the following questions:

- **Effectiveness on vulnerability finding**: Can SCENT find real TCP side-channel vulnerabilities from these two kernels? (§8.2)
- **Effectiveness of automated downscaling**: Does automated downscaling allow SCENT to cover more code and more importantly, find more vulnerabilities? (§8.3)
- **Effectiveness of model checking**: Does bounded model checking offer better scalability than bounded testing? (§8.4)

8.1 Evaluation setup

Implementation details. Our implementation of SCENT is built on a set of open-sourced program analysis platforms and tools. The static data-flow analysis used by Model Generator is built upon kernel-analyzer [46]. The concolic execution engine used by Model Transformer is built on top of KLEE [6] with the Z3 SMT solver [47]. The instrumentation is built on top of the LLVM compiler infrastructure [30]. The bounded model checking is done using the SPIN model checker [23].

Stub function abstraction and crafting a standalone system. As discussed in §5, we abstract a few functions to facilitate scalability and make the code amenable to model checking. The details are listed below. It initially takes us longer with Linux as we were finalizing the methodology. However, it took us only 2.5 weeks to build the model for FreeBSD afterwards.

- *Mutex and Lock related functions*: Use empty function (During model checking, TCP is executed as a single thread process).
- *Memory allocation*: Pre-allocate the memory and return the corresponding memory object (because the model checker cannot track dynamic memory).
- *Memory release*: Use empty function (because we preallocated the memory).
- *Callback functions*: Use empty functions (as limited by the state explosion issue, we only consider one interleaving of concurrent events; therefore, we can focus on TCP mechanisms).
- *Out of scope functions (IP layer or User space)*: Use empty function body or craft abstractions manually, to send packets with TCP layer information.
- *Functions that include assembly code*: Either replace with `glibc` functions or abstract them based on their logic (examples include printk or __memcpy). Since we need to keep the model deterministic, we replace prandom_u32 function with a fixed but arbitrarily chosen value.
- *Timer*: Return the fixed value captured from a snapshot based on a real connection (e.g., for `tcp_current_mss`). This helps eliminate the non-determinism in the model as well as the otherwise intractable state space (time as a new dimension) that we are not interested in.

In general, manually abstracting functions results in a risk of missing vulnerabilities (lowered true positives); however, this step is necessary to ensure the feasibility of model checking. If these manual abstractions cause false positives, they can be easily verified (in our experiments, we did not encounter such cases).

Testbed. We evaluated SCENT on two servers, each with a 2.6GHz (8-core) CPU and 128G memory. The secrets of interest are tabulated in Table 1. We consider the following scenarios: (1) the attacker has established its own TCP connection with the server; (2) the attacker sends packets to an open port at the server; and (3) the attacker sends packets to a closed port at the client. We assume that the attacker can either send packets to the client or the server, but not send to both[5]. We consider 6 different settings with regards to the victim's secret attributes and thus, with the three attacker scenarios, we run 36 experiments for each model. For the Linux model, we set 2 days as a hard limit for each experiment (given that it is more complex); while for the FreeBSD model, we set 1 day as a hard limit. We point out that these limits were imposed based on

[5]The latter cases can be handled by SCENT but we leave such evaluations for future work.

the computation capacity available, and to obtain results within a reasonable time frame. Based on the counter examples found by SCENT, we set up two virtual hosts (Debian OS with Linux kernel 4.8.0 and FreeBSD OS with kernel 13.0) to validate their veracity in real settings.

8.2 Discovered side-channels

Table 3 shows the violations found during our experiment. SCENT discovered a total of 53 distinct violations. Our manual verification confirmed that they are all true positives.

These violations relate to a total of 15 side-channels, of which 10 are found in Linux and 5 in FreeBSD. Here we define "a side-channel" as a branch that causes the violation. Since the same check over a shared variable can be applied at multiple branches, the same key constraint in Table 3 can be associated with multiple side-channels.

Five of the discovered side-channels (4,6,7,8,11) are based on shared variables that are not discovered before, namely, close port reset counter, tcp memory counter, and accept queue associate with Listen socket (details to follow). Seven side-channels (1,2,3,5,9,10,12) are new ways (i.e., execution path) to exploit known shared resources [1, 16, 48]. The remaining three side-channels (13,14,15) are known ways to exploit a known shared resource [7].

Based on the shared resources, the side-channels can be categorized into 4 classes. Next, we describe the details and provide an examplar to showcase in each case.

Reset counter based side-channels (Class A). Side-channel 1, 2, and 3 in Table 3 are caused by what is called the "open port RST packet rate," which is used to restrict sending too many RST packets from an open port. Side-channel 4 is caused by what is called the "close port RST packet rate," which is used to restrict sending too many RST packets from a closed port at a host.

Figure 7 shows how side-channel 1 can be exploited to infer the port number of a victim connection. During the guess phase, the attacker sends a spoofed ACK packet with a guessed port number. As shown in the left part of the figure, if the port number is the one used in the victim connection, the server will either accept or drop the packet (depending on the SEQ and ACK numbers); the response is sent to the client if appropriate. If the port number in the packet is not used (right part of the figure), the server determines that an ACK was received before any SYN packet. It therefore drops the packet but responds to the client with an OPENPORT RST packet. Because of this, the OPENPORT RST Counter is increased by one. Subsequently, in a check phase, the attacker will send 200 SYN-ACK packets to exhaust the OPENPORT RST limit (in 1 second) and observes the number of resposnes (RSTs) received from the server. If the attacker receives 200 RSTs, it means that the victim client is using that port number to communicate with server; else, it infers that the port number that it had guessed is incorrect.

SYN-backlog-based side-channels (Class B). The SYN backlog is a buffer that stores half-opened TCP sockets from connections during the three-way-handshake. Because the SYN backlog is associated with the "Listen" socket, its state is shared by all connections to the server. In order to prevent DoS attacks, the size of the SYN backlog is constrained to a shared limit. When the number of half-opened sockets has reached this limit, the SYN backlog buffer will either remove an old element or directly drop the current one (based on the OS kernel used, i.e., FreeBSD or Linux).

Side-channels 5 and 9 in Table 3, are caused because of this feature, exploiting which an attacker can infer the port number of a victim connection. Side-channels 10 and 11 can be used to infer the port number, SEQ number and ACK number; however it is practically hard to do so since the attacker needs to guess all three secrets simultaneously (which leads to a prohibitive search space).

To illustrate, let us consider the side-channel 9 as an example, which can be used to infer the port number of the victim connection. To begin with, the attacker establishes a number of half-opened sockets to just leave enough space for one additional spot in the SYN backlog buffer. Next, as shown in Figure 8, the attacker sends a spoofed SYN packet to server pretending to be the victim. If the guessed port number is already used in an established connection (i.e., the server and the client are communicating), the server will drop the SYN or send a challenge ACK, without allocating a new half-opened socket (as shown in the left part of the figure). Otherwise, a half-open socket is allocated and this makes the buffer full (as shown in the right part of the figure). Subsequently, the attacker sends a SYN packet with its own IP address towards creating a new half-opened socket, but more importantly to check whether the SYN backlog is full. Because Linux implements a LIFO (Last In First Out) algorithm to constrain the buffer size, if SYN backlog is full (as shown in the right part of the figure) , the server simply drops this new request for a half-opened socket without sending a response (assuming that SYN cookies are not enabled, which is common among quite a few cloud servers [2, 12]). Otherwise, the server will respond with a SYN-ACK to attacker.

Different from the Linux kernel, FreeBSD implements SYN backlog as a FIFO (First In First Out) buffer; this implies that an old half-opened socket will be dropped if the buffer is full. In this case, before sending the spoofed SYN packet, the attacker needs to *plant* its own half-opened socket first (via a legitimate SYN). After sending the spoofed SYN, it can infer whether buffer is full by checking if the previously *planted* half-opened socket still exists, by sending an ACK packet. Similar to the case with Linux, here we again assume that SYN cookies are not enabled.

TCP memory-counter-based side-channels (Class C). Side-channels 6, 7, and 8 are caused by a new shared variable discovered by SCENT. We refer to them as the TCP memory-counter-based side-channels. As shown in Table 3, all three vulnerabilities require an attacker to guess port number, SEQ number (in-window) and ACK number simultaneously, therefore they are not quite practical. Information leakage in this class are due to a global variable, viz., `tcp_memory_allocated`, which can be changed by any TCP connection. Table 3 depicts two key constraints associated with this variable: (a) `sysctl_tcp_mem[1]` indicates that currently the memory is under pressure, while (b) `sysctl_tcp_mem[2]` is used to indicate if the current allocated memory has reached a hard limit (thus, the server will drop data packets that need additional memory allocation). The different values of the above global variable can lead to different control flows, which in turn cause the server to send different packets to the attacker (in response to specific sequences of inputs). To exploit this feature, the attacker will first send a spoofed packet to try to change this global variable. The changes occur

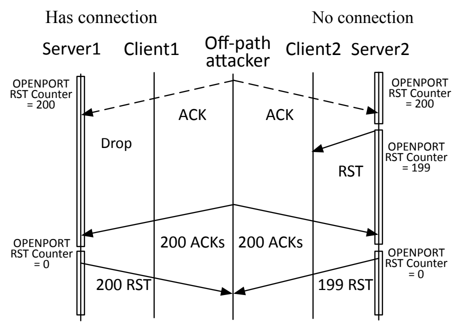

Figure 7: Reset counter based side channel example (Vulnerability 1, FreeBSD)

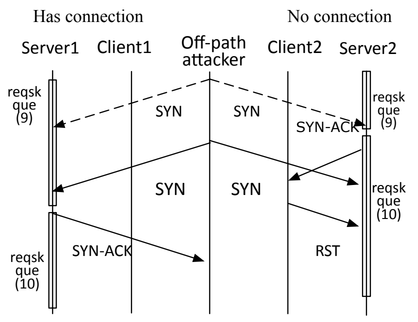

Figure 8: SYN-backlog based side channel example when SYN-Cookie is disabled (Vulnerability 9, Linux)

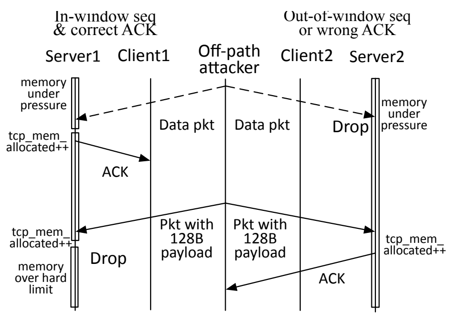

Figure 9: TCP memory counter based side channel example (Vulnerability 6, Linux)

Table 4: Branch Coverage Information Before and After Transformation

Kernel	Before Transformations		After Transformations		Increase Rate
	Num	Rate	Num	Rate	
Linux	476	36.62%	598	46.00%	25.63%
FreeBSD	618	33.59%	781	42.45%	26.38%

only when the secret attributes of interest (i.e., SEQ number, ACK number, and port number) are guessed correctly. Subsequently, the attacker sends its own packets to try to reach the aforementioned limit; it can observe if the global variable has changed, based on the patterns of packets that are received. A change indicates that its guess of the secret attributes holds true.

To showcase this class of side channels, we sketch an exemplary case study shown in Figure 9. First, the attacker subsumes (pre-allocates) a large volume of memory before the attack. Next, the attacker sends a spoofed long data packet with a guessed SEQ number and ACK number. If the SEQ number is in window and the ACK number is correct (as shown on the left), the long data payload is stored in a queue that holds out-of-order packets (packets that are in window but are not equal to the next expected packet i.e., `rcv_next`) causing an increase in the `tcp_memory_allocated` counter; otherwise, the server will simply drop the packet (as shown on the right). During a subsequent probing phase, the attacker deliberately sends an out-of-order packet with a large data payload on its own connection. This is designed to significantly increase `tcp_memory_allocated`. If `tcp_memory_allocated` has increased before (in the previous step) causing the server to reach its hard memory limit, it will cause a droppage of this packet; otherwise, the attacker will receive an ACK packet from server. Therefore, attacker can infer whether the guesssed secret attributes (SEQ number and ACK number) in the spoofed packet are correct or not.

Challenge counter based side channel (Class D). Side-channel 12 is a new one that is similar to previously reported old ones (13, 14, 15). Here, we explicitly include the challenge ack mechanism in Linux 3.8.0 towards validating previously reported side channels [7]; these are based on a global variable called `challenge_count` and have been extensively described in [7]. Furthermore, this has already been patched in Linux and other OSes.

8.3 Effectiveness of automated downscaling

Automated downscaling is the core innovation of SCENT that improves the code coverage of bounded model checking. In this subsection, we evaluate the effectiveness of this technique.

Table 4 shows the branch coverages achieved before and after the transformation of automated downscaling. The branch coverage rate was increased by 25.63% with regards to the Linux kernel and by 26.38% with the FreeBSD kernel. Although the final branch coverage rate is seemingly low at 46.00% (as in Linux model), during our manual analysis, we found that many of the uncovered branches were due to our limited input space. Specifically, we did not explore paths related to header options, paths that involve the server actively sending packets, paths that are related to connection termination before the "Closed" state, etc. If we discard these branches (which we do not expect to cover) the branch coverage rate improves to around 70%.

Besides code coverage, a more important question is whether automated downscaling enables SCENT to discover more side-channels. The second last column in Table 3 shows the answer to this question. In fact, *none of the side-channels can be found without automated downscaling* (all require it). We believe that this highlights the importance and effectiveness of our technique.

8.4 Performance of model checking

One important design choice we made when building SCENT is whether to use bounded testing [32], wherein we can directly test an unmodified kernel, or use bounded model checking. The benefit of model checking is that it will visit each state only once, thereby avoiding the execution of redundant steps and improving the performance of testing. In this subsection, we compare the performance of bounded model checking with bounded testing, in terms of number of iterations. Figure 10 shows the result. Basically, bounded model checking executes 4 orders of magnitude fewer iterations than blind enumeration (i.e., bounded testing).

The next choice we made, that is related to the performance of model checking, is imposing a limit on the number of packets to be enumerated during bounded model checking. Figure 11 shows how the time of one round of model checking increases as the number of packets increases. Figure 12 shows how the memory usage of the model checker increases as the number of packets increases. When

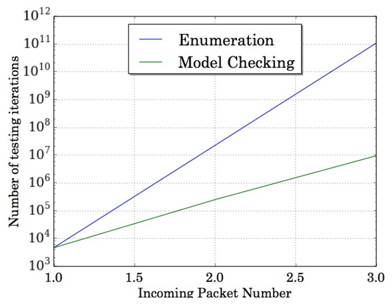

Figure 10: Number of testing iterations versus number of incoming packets

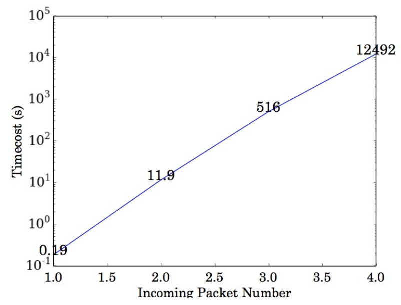

Figure 11: Timecost of one model checking run versus number of incoming packets

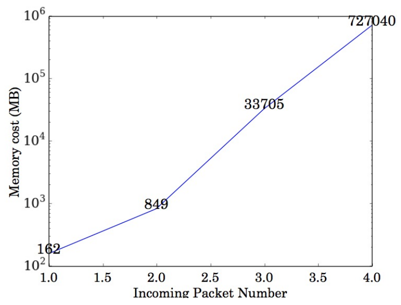

Figure 12: Memory cost of one model checking run versus number of incoming packets

the number of packets increase to 4, it will either take too long to test all the different configurations or exhaust all the memory on the testbed.

9 CASE STUDY

When the port number is leaked, an attacker can infer whether the victim client is communicating with the server (either during the three way handshake or in ESTABLISHED state). This leaks the victim user's privacy. Side-channels 1, 3, 4, 5, 9 can leak port number information, and can therefore be used to achieve this attack. In the previous section, we discussed how such an attack can be launched. We now construct a real attack to demonstrate the impact of the corresponding side-channels found by SCENT.

As an exemplar, we pick side-channel 1 (as shown in Figure 7), and evaluate it in terms of metrics such as success rate and the time to succeed. In our experiment, we used a Ubuntu 14.04 host on a university campus as the victim client. The victim server is a virtual machine running FreeBSD OS from a different Ubuntu 14.04 host. The attack machine is a Ubuntu 16.04 host on the same campus. The steps in the attack process are listed below:

(1) Synchronize machine times between attacker and server;
(2) Send spoofed and unspoofed ACK packets to linearly guess a port number range based on the number of RST packets received;
(3) Given a port number range, use binary search to locate the specific port number.

The attacker can guess 200 different port numbers (via spoofed packets) in one second; otherwise, spoofed packets will always reach the reset counter limit. The attacker can guess the port number starting from the Ephemeral port range [15], and then guess the remainder of the port range. Our experiment shows that this attack of inferring a correct port number is achievable within an average time of 73 seconds with a 100% success rate.

10 RELATED WORK

TCP side channel attacks. In the last decade, several TCP side channels have been manually found by researchers. These side channels can be utilized to (1) cause a TCP inference attack [7, 11, 18, 37, 38], which in turn can lead to a hijack of the connection and injection of malicious data; (2) measure host attributes without exposing the attacker's IP address (examples include performing an idle port scan [16] or measuring the RTT between two hosts [1]). Roughly these distinct attacks can be mapped onto the exploitation of four categories of side-channel vulnerabilities: (1) Shared rate limit: these side-channels relate to a rate limit that is shared across the victim and an off-path attacker connection, such as IPID counter [4, 10, 18, 35, 39, 49], the challenge ACK rate limit [7], the reset rate limit and the shared SYN backlog queue limit [1, 16]. (2) System-wide packet counter: As the name suggests a packet counter is shared globally in these cases [37, 38]. (3) Wireless link: Wireless contention results in information leakage in these cases [11] (timing-based side channel). (4) Browser implementation's feature: A per destination port-counter and a FIFO HTTP request queue cause information leaks [19].

While most of these side-channels are discovered manual by domain experts, SCENT aims to automate the discovery in a principled way. Our evaluation shows that SCENT indeed can detected (both new and known) side-channels.

Side channel detection. Most previous side-channel vulnerabilities have been discovered manually (e.g., using domain expertise). However, a few side-channel detection tools have been proposed. [9] uses static taint analysis to discover system-wide TCP packet counter side-channel vulnerabilities. Generally, static taint analysis can be guaranteed to find all true violations, but suffers from high false positives. By relying on violation of the non-interference property, SCENT can avoid high false positives and can detect side-channels caused by different shared variables. There are also several efforts relating to the detection of other types of side-channels but these are orthogonal to our work [5, 44].

Program analysis and testing. There are also several efforts that use program analysis (e.g., static and/or dynamic analysis) to find bugs or other types of attacks in TCP implementations [26, 28]. These are orthogonal to our work and address significantly different problems.

Model checking and formal verification have been used to analyze the robustness of TCP implementations [17, 33]; however, their objectives are signficiantly different. More importantly, SCENT uses automated downscaling to improve the effectivenss of model checking.

Besides bounded module checking [14, 29, 31], one could also do bounded testing [32]. The advantage of bounded testing is that it does not require additional modification to the target program, while model checking usually requires generating a model amenable to the model checker. However, as shown in our evaluations, by avoiding redundant states, a model checker can help explore a larger input space.

Program transformation has been used to assist testing using fuzzing to patch hard-to-flip branches (like checksum checks) as a way of improving code coverage [27, 36, 45]. In contrast, SCENT tries to coerce both true and false path to be visited and most of the target branches (Table 3 column 3) have simple constraints. In addition, SCENT changes the internal states instead of "disabling" the branch.

11 CONCLUSIONS

In this paper, we consider the challenging problem of developing a principled approach to discovering hard-to-find TCP side-channels. We use model checking as a basis for finding violations of the non-interference property between simultaneous TCP connections, which we argue is a precursor to exploitable side channels. As our main contribution, we build a tool SCENT that achieves our goal by addressing two hard challenges in making model checking amenable to our goal namely, (a) making a TCP code base self contained after pruning irrelevant parts and (b) systematically downscaling both the input space and the model state space by means of principled program transformations. We use the counter-examples generated by the transformed model checker in SCENT to discover 12 new side channels and also validate all previously discovered ones. In this work, we limit ourselves to side channels that facilitate the inference of a specific set of secret attributes (e.g., SEQ number); we will expand our threat model to find other types of vulnerabilities (e.g., idle port scans) and with more scenarios (e.g., attacker can send packets to both the client and the server and/or with different OSes) in the future.

ACKNOWLEDGMENTS

This research was partially sponsored by the U.S. Army Combat Capabilities Development Command Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Combat Capabilities Development Command Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on. It was also partially supported by the NSF grant 1652954.

REFERENCES

[1] Geoffrey Alexander and Jedidiah R Crandall. 2015. Off-path round trip time measurement via TCP/IP side channels. In *2015 IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 1589–1597.

[2] Apache Geode Documentation [n. d.]. Disable TCP SYN Cookies. Retrieved May 15, 2019 from https://geode.apache.org/docs/guide/14/managing/monitor_tune/disabling_tcp_syn_cookies.html

[3] Thomas Ball and Sriram K. Rajamani. 2001. The SLAM Toolkit. In *Computer Aided Verification*, Gérard Berry, Hubert Comon, and Alain Finkel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 260–264.

[4] Steven M Bellovin. 2002. A technique for counting NATted hosts. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*. ACM, 267–272.

[5] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut Kandemir. 2019. CaSym: Cache aware symbolic execution for side channel detection and mitigation. In *CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation*. IEEE, 0.

[6] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *OSDI*, Vol. 8. 209–224.

[7] Yue Cao, Zhiyun Qian, Zhongjie Wang, Tuan Dao, Srikanth V. Krishnamurthy, and Lisa M. Marvel. 2016. Off-Path TCP Exploits: Global Rate Limit Considered Dangerous. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 209–225. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/cao

[8] Martim Carbone, Weidong Cui, Long Lu, Wenke Lee, Marcus Peinado, and Xuxian Jiang. 2009. Mapping kernel objects to enable systematic integrity checking. In *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 555–565.

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

[10] Weifeng Chen, Yong Huang, Bruno F Ribeiro, Kyoungwon Suh, Honggang Zhang, Edmundo de Souza e Silva, Jim Kurose, and Don Towsley. 2005. Exploiting the IPID field to infer network path and end-system characteristics. In *International Workshop on Passive and Active Network Measurement*. Springer, 108–120.

[11] Weiteng Chen and Zhiyun Qian. 2018. Off-Path {TCP} Exploit: How Wireless Routers Can Jeopardize Your Secrets. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 1581–1598.

[12] Cisco [n. d.]. Defenses Against TCP SYN Flooding Attacks. Retrieved May 15, 2019 from https://www.cisco.com/c/en/us/about/press/internet-protocol-journal/back-issues/table-contents-34/syn-flooding-attacks.html

[13] Edmund M. Clarke and E. Allen Emerson. 1982. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logics of Programs*, Dexter Kozen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 52–71.

[14] Lucas Cordeiro, Jeremy Morse, Denis Nicole, and Bernd Fischer. 2012. Context-bounded model checking with ESBMC 1.17. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 534–537.

[15] Cymru [n. d.]. Ephemeral Source Port Selection Strategies. Retrieved May 15, 2019 from https://www.cymru.com/jtk/misc/ephemeralports.html

[16] Roya Ensafi, Jong Chun Park, Deepak Kapur, and Jedidiah R. Crandall. 2010. Idle Port Scanning and Non-interference Analysis of Network Protocol Stacks Using Model Checking. In *Proceedings of the 19th USENIX Conference on Security (USENIX Security'10)*. USENIX Association, Berkeley, CA, USA, 17–17. http://dl.acm.org/citation.cfm?id=1929820.1929843

[17] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. 2016. Combining model learning and model checking to analyze TCP implementations. In *International Conference on Computer Aided Verification*. Springer, 454–471.

[18] Yossi Gilad and Amir Herzberg. 2012. Off-Path Attacking the Web.. In *WOOT*. 41–52.

[19] Yossi Gilad and Amir Herzberg. 2013. When tolerance causes weakness: the case of injection-friendly browsers. In *Proceedings of the 22nd international conference on World Wide Web*. ACM, 435–446.

[20] Patrice Godefroid. 1997. Model Checking for Programming Languages Using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*. ACM, New York, NY, USA, 174–186. https://doi.org/10.1145/263699.263717

[21] J. A. Goguen and J. Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy*. 11–11. https://doi.org/10.1109/SP.1982.10014

[22] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2003. Software Verification with BLAST. In *Proceedings of the 10th International Conference on Model Checking Software (SPIN'03)*. Springer-Verlag, Berlin, Heidelberg, 235–239. http://dl.acm.org/citation.cfm?id=1767111.1767128

[23] Gerard J. Holzmann. 1997. The model checker SPIN. *IEEE Transactions on software engineering* 23, 5 (1997), 279–295.

[24] Gerard J. Holzmann and Rajeev Joshi. 2004. Model-Driven Software Verification. In *Model Checking Software*, Susanne Graf and Laurent Mounier (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 76–91.

[25] G. J. Holzmann and M. H. Smith. 1999. A practical method for verifying event-driven software. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*. 597–607. https://doi.org/10.1145/302405.302710

[26] Samuel Jero, Endadul Hoque, David Choffnes, Alan Mislove, and Cristina Nita-Rotaru. 2018. Automated attack discovery in TCP congestion control using a model-guided approach. In *Proceedings of NDSS*.

[27] Ulf Kargén and Nahid Shahmehri. 2015. Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 782–792.

[28] Nupur Kothari, Ratul Mahajan, Todd Millstein, Ramesh Govindan, and Madanlal Musuvathi. 2011. Finding protocol manipulation attacks. In *ACM SIGCOMM computer communication review*, Vol. 41. ACM, 26–37.

[29] Daniel Kroening and Michael Tautschnig. 2014. CBMC–C bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 389–391.

[30] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[31] Florian Merz, Stephan Falke, and Carsten Sinz. 2012. LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In *International Conference on Verified Software: Tools, Theories, Experiments*. Springer, 146–161.

[32] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding crash-consistency bugs with bounded black-box crash testing. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 33–50.

[33] Madanlal Musuvathi and Dawson R. Engler. 2004. Model Checking Large Network Protocol Implementations. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1 (NSDI'04)*. USENIX Association, Berkeley, CA, USA, 12–12. http://dl.acm.org/citation.cfm?id=1251175.1251187

[34] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. 2002. CMC: A Pragmatic Approach to Model Checking Real Code. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2002), 75–88. https://doi.org/10.1145/844128.844136

[35] Paul Pearce, Roya Ensafi, Frank Li, Nick Feamster, and Vern Paxson. 2017. Augur: Internet-wide detection of connectivity disruptions. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 427–443.

[36] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 697–710.

[37] Zhiyun Qian and Z Morley Mao. 2012. Off-path TCP sequence number inference attack-how firewall middleboxes reduce security. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 347–361.

[38] Zhiyun Qian, Z Morley Mao, and Yinglian Xie. 2012. Collaborative TCP sequence number inference attack: how to crack sequence number under a second. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 593–604.

[39] Zhiyun Qian, Z Morley Mao, Yinglian Xie, and Fang Yu. 2010. Investigation of triangular spamming: A stealthy and efficient spamming technique. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 207–222.

[40] Jean-Pierre Queille and Joseph Sifakis. 1982. Specification and Verification of Concurrent Systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*. Springer-Verlag, London, UK, UK, 337–351. http://dl.acm.org/citation.cfm?id=647325.721668

[41] SCENT [n. d.]. SCENT: TCP Side Channel Excavation Tool. https://github.com/seclab-ucr/SCENT

[42] Kevin Sullivan, Jinlin Yang, David Coppit, Sarfraz Khurshid, and Daniel Jackson. 2004. Software assurance by bounded exhaustive testing. In *ACM SIGSOFT Software Engineering Notes*, Vol. 29. ACM, 133–142.

[43] Rijnard van Tonder, John Kotheimer, and Claire Le Goues. 2018. Semantic crash bucketing. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*.

[44] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. 2017. CacheD: Identifying cache-based timing channels in production software. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. 235–252.

[45] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 497–512.

[46] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. 2012. Improving Integer Security for Systems with {KINT}. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 163–177.

[47] Z3Prover/z3 [n. d.]. The Z3 Theorem Prover. Retrieved May 4, 2019 from https://github.com/Z3Prover/z3

[48] Xu Zhang, Jeffrey Knockel, and Jedidiah R Crandall. 2015. Original SYN: Finding machines hidden behind firewalls. In *2015 IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 720–728.

[49] Xu Zhang, Jeffrey Knockel, and Jedidiah R Crandall. 2018. ONIS: Inferring TCP/IP-based Trust Relationships Completely Off-Path. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2069–2077.