A Formal Analysis of SCTP: Attack Synthesis and Patch Verification

Jacob Ginesin*
ginesin.j@northeastern.edu
Northeastern University

Max von Hippel*
vonhippel.m@northeastern.edu
Northeastern University

Evan Defloor†
defloor.e@northeastern.edu
Northeastern University

Cristina Nita-Rotaru†
c.nitarotaru@northeastern.edu
Northeastern University

Michael Tüxen†
tuexen@fh-muenster.de
FH Münster

Abstract

SCTP is a transport protocol offering features such as multi-homing, multi-streaming, and message-oriented delivery. Its two main implementations were subjected to conformance tests using the PACKETDRILL tool. Conformance testing is not exhaustive and a recent vulnerability (CVE-2021-3772) showed SCTP is not immune to attacks. Changes addressing the vulnerability were implemented, but the question remains whether other flaws might persist in the protocol design.

We study the security of the SCTP design, taking a rigorous approach rooted in formal methods. We create a formal PROMELA model of SCTP, and define ten properties capturing the essential protocol functionality based on its RFC specification and consultation with the lead RFC author. Then we show using the SPIN model checker that our model satisfies these properties. We next define four representative attacker models – Off-Path, where the attacker is an outsider that can spoof the port and IP of a peer; Evil-Server, where the attacker is a malicious peer; Replay, where an attacker can capture and replay, but not modify, packets; and On-Path, where the attacker controls the channel between peers. We modify an attack synthesis tool designed for transport protocols, KORG, to support our SCTP model and four attacker models.

We synthesize fourteen unique attacks using the attacker models – including the vulnerability reported in CVE-2021-3772 in the Off-Path attacker model, four attacks in the Evil-Server attacker model, an opportunistic `ABORT` attack in the Replay attacker model, and eight connection manipulation attacks in the On-Path attacker model. We show that the proposed patch eliminates the vulnerability and does not introduce new ones according to our model and protocol properties. Finally, we identify and analyze an ambiguity in the SCTP RFC. Using the ALLENNLP coreference resolution model, we show that the ambiguous text could be reasonably interpreted in two ways; then we model the incorrect interpretation, and synthesize a novel attack against it. To avoid novice implementers incorrectly interpreting the RFC, we propose an erratum, and using the same ALLENNLP model, we show that it eliminates the ambiguity.

*Contributed equally.
†Listed alphabetically.

1 Introduction

Transport protocols play a crucial role in transmitting data across the Internet either directly – as in UDP [3] and DCCP [21], which provide unreliable communication, and TCP [19] and SCTP [57], which provide reliable communication – or by supporting secure protocols – e.g., UDP supports DTLS [50] and QUIC, while TCP supports TLS [49]. Thus, it is critical that transport protocols are designed and implemented to be bug-free and secure.

SCTP is a transport layer protocol proposed as an alternative to TCP, offering new features, such as multi-homing, multi-streaming, and message-oriented delivery. Among other use-cases, it is the data channel for WebRTC [5], which is used by such applications as Facebook Messenger [33], Microsoft Teams [36], and Discord [62]. The design of SCTP is described in RFC documents, the most recent one being RFC 9260 [57], and implemented in Linux [2] and FreeBSD [4]. These implementations were tested using PACKETDRILL [1, 14] and analyzed with WIRESHARK [51]. Some limited efforts also analyzed the SCTP design using formal methods. The works in [59, 60] focused only on bugs and did not consider attacks, while the work in [52] focused on attacks, but modeled only limited aspects of connection establishment to compare the resilience of SCTP and TCP to SYN-FLOOD attacks. A recent vulnerability – CVE-2021-3772 [48] – shows the importance of conducting a much more comprehensive formal analysis. Although a patch was proposed in RFC 9260 [57], and adapted by FreeBSD, the question remains whether other flaws might persist in the protocol design and whether the patch might have introduced additional vulnerabilities. To the best of our knowledge, no prior works formally analyzed the entire SCTP connection establishment and teardown routines in a security context.

In this work, we take an approach rooted in formal methods to study the security of SCTP. Our approach is based on *attack*

synthesis, where the goal is, given a program that behaves correctly, and an attacker model, to find an attack that can lead the program to behave incorrectly.[1] Combined with other formal methods, such as model checking, this approach allows us to precisely study the behaviors of protocols such as SCTP under different threat models.

Model Design and Verification. We start by creating a finite state machine (FSM) model for the SCTP design as specified in RFCs 4960 [54] and 9260 [57], and writing ten properties the models should satisfy based on a close reading of the RFC documents and discussions with the lead SCTP author [2]. Our properties are defined in Linear Temporal Logic (LTL) and characterize the standard establishment and teardown routines, the proper functioning of the cookie timer, and the fact that SCTP does not support half-open connections. Using the SPIN model checker, we automatically verify that our SCTP model meets these properties (behaves correctly) when not under attack.

Attack Synthesis. Next we define four attacker models (Off-Path, Evil-Server, Replay, and On-Path), which are representative for transport protocols and provide a wide range of attacker capabilities allowing us to understand the behavior of SCTP when under attack. The Off-Path attacker model describes an attacker who may or may not know the IP address or port of either peer, but cannot read the messages in-transit, and does not know the authentication secrets (which in SCTP are called the "vtags") of the association. Thus, its injected messages should theoretically be ignored. In the Evil-Server attacker model, one peer in an association is malicious, and aims to guide the other peer into some vulnerable state. The Replay attacker model describes an attacker capable of capturing messages from the communication channel and replaying them without modification. In the On-Path attacker model, the attacker controls the channel connecting the peers, and can intercept, drop, and inject authenticated messages at-will.

We use an attack synthesis tool for transport protocols called KORG, based on LTL model-checking [64]. We automatically synthesize attacks against our SCTP model, for each LTL property and attacker model. In the Off-Path case, we automatically find the attack from CVE-2021-3772. We find numerous attacks in the Evil-Server and On-Path attacker models, e.g., an Evil-Server attack that establishes a connection with the victim peer and then leaves it stranded, and an On-Path attack that injects messages guiding peers into Shutdown_Received (an illegal passive/passive teardown). These results highlight the importance of implementation level defenses against an Evil-Server, and an end-to-end security model to prevent On-Path attacks. We also find one Replay attack, highlighting the security-criticality of the transmission sequence number (TSN).

Patch Verification. We next configure the model to include the patch introduced in RFC 9260 [57] and show that the patch fixes the problem, i.e. the property that was violated by the attack is now met under the attacker model where that attack was discovered. We further show that in all other attacker models, the same attacks exist with or without the patch, so no new vulnerabilities are introduced by the patch. KORG is sound and complete, meaning that (1) if it finds an attack then the attack is real (against the model), and (2) if any attacks against the model and property exist, of the type KORG looks for, then given sufficient time and memory, KORG will find one [64]. Since the CVE attack is the kind of attack KORG looks for using the Off-Path attacker model, the fact that we find the attack when the patch is disabled, but find no such attacks when it is enabled, suffices to prove that the attack does not exist in the patched model.

RFC Disambiguation. Motivated by the fact that CVE-2021-3772 was caused by a lack of clarity in RFC 4960, we carefully analyze the RFCs for ambiguities. We identify a portion of RFC 9260 that seems ambiguous to us, and confirm using the ALLENNLP coreference resolution [23] natural language processing (NLP) model that the text can in fact be interpreted in two ways. We confirm which is correct by consulting with the lead SCTP RFC author; then model the incorrect interpretation and, using the SPIN [28] model-checker, show that it is vulnerable to a potentially serious attack where the attacker can trick a peer in an association into using the wrong vtag. We propose an RFC erratum and show using the same NLP approach, that it unambiguously communicates the correct interpretation. Finally, we use PACKETDRILL [14] to confirm that the Linux and FreeBSD implementations interpret the ambiguous text correctly. Note, the FreeBSD implementation was co-authored by the lead SCTP RFC author, so naturally it interprets the RFC correctly.

Contributions. We summarize our contributions:

- *Model:* We model the original SCTP RFC [54] using PROMELA. Our model can be configured with or without the CVE patch from RFC 9260 [57]. It is endorsed by the lead SCTP RFC author and faithfully captures the SCTP connection and teardown routines, including the exchange of messages, the user-on-the-loop and its commands, and the handling out-of-the-blue packets.
- *Verification:* We formalize ten novel correctness properties for SCTP in LTL based on a close reading of the RFCs and use SPIN to prove that our model satisfies all ten when no attacker is present.
- *Attack Synthesis:* We introduce four attacker models for SCTP. Then we modify KORG to support *packets* and *replay attacks*, and use it to synthesize attacks in the context of each attacker model. For Off-Path, we rediscover the CVE before the patch was applied, but not after. For Evil-Server, we find four attacks that, depending on implementation details, could

[1] This is totally different from program synthesis, where the problem is, given some property, to conjure a program that satisfies it.

[2] We do not seek to construct a complete set of properties, as we're interested in studying the security-relevant behaviors of SCTP rather than creating an all-encompassing specification. Also, defining a complete specification in LTL is impractical, as LTL is optimized for efficient model checking.

leave a victim peer deadlocked or stranded in some liveness cycle, unable to automatically de-associate. For Replay, we find one attack that, depending on the security of the TSN, could prevent two peers from establishing a connection. We find six similar On-Path attacks where the attacker leads the peers into some illegal state or cycle, violating a property.

• *Patch Verification:* We show that the patch fixes the problem, i.e. the property that was violated by the attack is now met under the attacker model wherein the CVE attack was discovered. Moreover, we show that no new attacks are made possible by the patch in any of the attacker models against any of our ten properties.

• *RFC Disambiguation:* We identify an ambiguity in RFC 9260 which, we show, could be reasonably misinterpreted in a way that opens the protocol to a new vulnerability. We confirm that neither implementation makes the mistake, and to avoid it in future implementations, we suggest an RFC erratum which we show to be unambiguous.

Ethics. We disclosed all of our results to the chair of the SCTP RFC committee.

Code. All of our results are reproducible with our open source code, available at https://github.com/sctpfm.

2 SCTP

In this section we overview SCTP and previous efforts to validate it, as well as our approach to analyzing its security.

2.1 Overview

SCTP is a transport protocol proposed as an alternative to TCP, offering enhanced performance, security features, and greater flexibility. It is specified in several RFCs, each introducing significant modifications. RFC 9260 [57], which obsolesced RFC 4960 [54], made numerous small clarifications and improvements, including a critical patch for CVE-2021-3772. On the other hand, RFC 4960, which obsolesced the original specification in RFC 2960 [58], introduced major structural changes to the protocol as described in the errata RFC 4460 [15]. SCTP is implemented in Linux [2] and FreeBSD [4].

SCTP is a two peer protocol where each peer runs the same state machine. However, during connection establishment, the two peers play different roles – while one peer progresses through the *active* routine in the state machine, the other peer must take a corresponding sequence of *passive* transitions.[3] For teardown there are two options: graceful or graceless. During graceful tear-down, one peer can act actively and the other passively, or they can both take an active role. Graceless teardown happens in a single step.

Peer States. An SCTP peer is identified by a set of IP addresses and a port number. At any given time, each peer exists in one of finitely many states: Closed, in which no association exists; Cookie_Wait and Cookie_Echoed, used during active establishment; Established, in which an association exists and data can be transferred; Shutdown_Received and Shutdown_Ack_Sent, used by the passive peer during teardown; and Shutdown_Pending and Shutdown_Sent, used by the active peer during teardown. In active/active teardown, both peers use Shutdown_Ack_Sent.

Packets. An SCTP packet consists of a common header and a number of chunks. An essential component of the connection establishment design is authentication of packets between the peers using a random integer called the *verification tag*, or vtag, which is initialized using an *initiate tag*, or itag, during establishment. The packet header contains the source and destination port number, vtag, and a checksum. The chunk types are `INIT`, `INIT_ACK`, `COOKIE_ECHO`, and `COOKIE_ACK`, used during establishment; `DATA` and `DATA_ACK`, used for data transmission once an association has been established; `ERROR`, used to communicate when an error has occurred; `SHUTDOWN`, `SHUTDOWN_ACK`, and `SHUTDOWN_COMPLETE`, used during graceful teardown; `ABORT`, used during graceless teardown; and `HEARTBEAT` and `HEARTBEAT_ACK`, used for crash detection. Chunks contain parameters, e.g., `INIT` and `INIT_ACK` chunks (but no others) contain an itag, and (only) `INIT_ACK` chunks contain a *state cookie*, which includes a message authentication code, a timestamp indicating when the cookie was created, and a cookie lifespan. There are various kinds of `ERROR` chunks, each indicating a different error condition, e.g., `COOKIE_ERROR` which indicates receipt of a valid but expired state cookie. Much like TCP, SCTP uses sequence numbers, called Transmission Sequence Numbers (TSNs). The initial TSN in an association is proposed by an active participant in the connection establishment routine, and is incremented with each data transmission thereafter.

Connection Establishment. In active/passive establishment (Figure 1), the active peer sends a packet with an `INIT` chunk, containing a nonzero random itag. For the remainder of the association, this (active) peer will only accept packets from the passive peer that contain a vtag equal to the itag in the `INIT` it sent. The passive peer replies with a packet containing an `INIT_ACK` chunk, which also contains a nonzero random itag. For the remainder of the association, the passive peer will only accept packets which contain this itag value as the vtag in the common header. By checking the vtag, each peer protects itself from processing packets sent by an attacker not knowing the recipient's vtag.

Connection Teardown. Teardown can occur gracefully, via the active/passive or active/active routines, or gracelessly, with an `ABORT`. During active/passive teardown (Figure 2), the active peer sends a `SHUTDOWN` chunk, to which the passive peer responds with `SHUTDOWN_ACK`. The active peer then sends `SHUTDOWN_COMPLETE` and both transition to Closed. Active/active teardown is also possible, in which the peers exchange, in the following order: `SHUTDOWN`, `SHUTDOWN_ACK`,

[3] SCTP also supports an initialization routine where both peers are active, called "initialization collision". However, this routine is described in the RFC as an edge-case, rather than an intended use-case.

and SHUTDOWN_COMPLETE messages. The third option is that a peer can gracelessly abort a connection by sending an ABORT chunk. In this case, both peers immediately transition to Closed. Once the association is closed, the vtags are forgotten, and when either peer enters a new association, it will randomly choose a new itag (to become its vtag).

Timers. The SCTP connection routines use three timers: Init, Cookie, and Shutdown. The goal of the Init Timer is to stop the active peer in an establishment routine from getting stuck waiting forever for the passive peer to respond to its INIT with an INIT_ACK. The goal of the Cookie Timer is similar: it stops that same active peer from getting stuck waiting forever for the passive peer to respond to its COOKIE_ECHO. The Shutdown Timer plays a similar role but in the teardown routine, stopping the active peer in teardown from getting stuck waiting for a SHUTDOWN_ACK.

Out-of-the-Blue Packet Handling. In SCTP a message is considered *out-of-the-blue* (OOTB) if the recipient cannot determine to which association the message belongs, i.e., if it has an incorrect vtag, or is an INIT with a zero-valued itag. Specifically, an OOTB message will be discarded if: 1) it was not sent from a unicast IP, 2) it is an ABORT with an incorrect vtag, 3) it is an INIT with a zero itag or incorrect vtag[4], 4) it is a COOKIE_ECHO, SHUTDOWN_COMPLETE, or COOKIE_ERROR, and is either unexpected in the current state or has an incorrect vtag, or 5) it has a zero itag or incorrect vtag.

Unexpected Packet Handling. A message is *unexpected* if it is not OOTB, but nevertheless, the recipient does not expect it. SCTP handles unexpected packets as described in Algr. 1.

Other Functionality. Other functionalities of SCTP include a "tie-tag" nonce mechanism used to authenticate a reconnecting peer after a restart; congestion control[5]; fragmentation and reassembly of DATA chunks; chunk bundling; support for the Internet Control Message Protocol (ICMP); and multihoming. We do not consider this functionality in our analysis, and we refer the reader to [57] for more details.

2.2 Prior Validation

Conformance testing. The Linux and FreeBSD implementations were tested with PACKETDRILL [1] and fuzz-tests, suggesting they are crash-free and follow the RFCs. But this does not necessarily imply the *design* in the RFCs behaves correctly in the (a) absense or (b) presense of an attacker.

Formal analysis. For (a), some prior works formally analyzed SCTP using Colored Petri Net models [39, 59, 60, 65] in CPNTOOLS. This software can check for livelocks (i.e. liveness violations) and deadlocks (stuck states), but it cannot model-check arbitrary logical properties, which seriously limits the use-cases for such models. One prior work studied (b),

[4]Per RFC 4960, respond with an ABORT having the vtag of the current association. But per RFC 9260, discard it.

[5](based on TCP congestion control)

Algorithm 1 Unexpected Packet Handling

```
Require: Unexpected msg
  if msg.chunk = INIT then
    if state = Cookie_Wait or msg does not indicate new addresses added then
      Send INIT_ACK with vtag = msg.itag
    else
      Discard msg and send ABORT with vtag = msg.itag
    end if
  else if msg.chunk = COOKIE_ECHO then
    if msg.timestamp is expired then
      Send COOKIE_ERROR
    else if msg has fresh parameters then
      Form a new association
    else
      /* initialization collision */
      Set vtag = msg.vtag
      goto Established
    end if
  else if msg.chunk = SHUTDOWN_ACK then
    Send SHUTDOWN_COMPLETE with vtag = msg.vtag
  else
    Discard msg
  end if
```

modeling the four-way handshake used by SCTP and comparing it to the three-way handshake used by TCP in the presence of an attacker, with the Uppaal model-checker [52]. However, the model is closed-source and does not include the teardown routine. It is unclear whether the model includes OOTB or unexpected packet handling. We summarize the differences between these prior models and our own in Table 1. Finally, the IETF published a security memo for SCTP, but it is not a comprehensive analysis, rather, it simply summarizes prior conversations about security from the SCTP user-group [55].

CVE-2021-3772 attack and patch. As reported in CVE-2021-3772 [48], the prior version of SCTP specified in RFCs 2960 [58] and 4960 [54] is vulnerable to a denial-of-service attack. The reported vulnerability worked as follows. Suppose SCTP peers A and B have established a connection and an off-channel attacker knows the IP addresses and ports of the two peers, but not the vtags of their existing connection. The attacker spoofs B and sends a packet containing an INIT to A. The attacker uses a zero vtag as required for packets containing an INIT. The attacker must use an illegal parameter in the INIT, e.g., a zero itag.

Peer A, having already established a connection, treats the packet as out-of-the-blue, per RFC 2960 §8.4 and 5.1, which specify that as an association was established, A should respond to the INIT containing illegal parameters with an ABORT and go to Closed. But in RFCs 2960 and 4960, it is unspecified which vtag should be used in the ABORT. Some implementations used the expected vtag, which is where a vulnerability

Work	RFC	Open-Source	Establish	Teardown	OOTB	Unexpected	Livelocks	Deadlocks	Properties
Martins et. al. [39]	2960	N	Y	Y	N	N	Y	Y	N
Blanchet et. al. [65]	2960	N	Y	Y	N	N	Y	Y	N
Vanit-Anunchai [59]	4960	N	Y	Y	Y	Y	Y	Y	N
Vanit-Anunchai [60]	4960	N	Y	Y	Y	Y	Y	Y	N
Saini and Fehnker [52]	4960	N	Y	N	N	N	Y	Y	Y
Ours	4960 & 9260	Y	Y	Y	Y	Y	Y	Y	Y

Table 1: Prior formal SCTP analyses versus ours. RFC column reports modeled version, and open-source column reports whether the model is open-source. The remaining columns report whether the model includes the establish and teardown routines, OOTB logic, or unexpected packet handling; and if it can be used to check for livelocks or deadlocks, or to verify arbitrary properties.

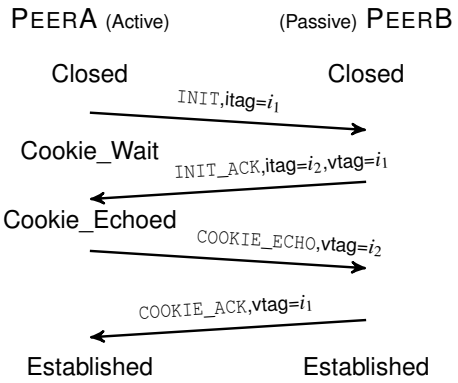

Figure 1: Message sequence chart illustrating SCTP active/-passive association establishment routine. Arrows indicate communication direction and time flows from the top down.

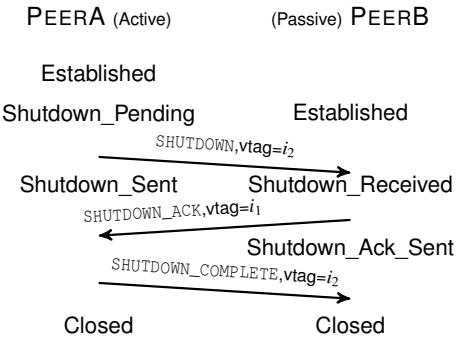

Figure 2: SCTP active/passive association teardown.

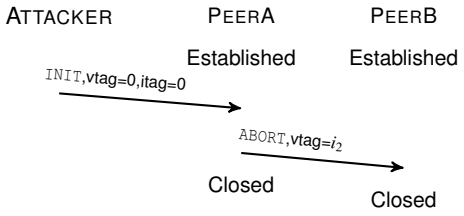

Figure 3: Attack disclosed in CVE-2021-3772. Peers A and B begin having established an association with vtags i_1, i_2 (resp.). The Attacker transmits an invalid `INIT` chunk to A, spoofing the port and IP of B. Peer A responds by sending a valid `ABORT` to B, which closes the association. By sending a single invalid `INIT` the Attacker performs a DoS.

arises. Since the attacker spoofed the IP and port of Peer B, Peer A sends the `ABORT` to Peer B, not the attacker. When Peer B receives the `ABORT`, it sees the correct vtag, and tears down the connection. Thus, by injecting a single packet with zero-valued tags, the attacker tears down the connection, pulling off a DoS. The attack is illustrated in Figure 3.

RFC 9260 patches CVE-2021-3772 using a strict defensive measure, wherein OOTB `INIT` packets with empty or zero itags are discarded, without response. FreeBSD [4] uses this patch. Linux, on the other hand, adopts a different patch [38], wherein the peer receiving the `ABORT` with the zero vtag simply ignores it (rather than close the connection).

3 Our SCTP Model

In this section, we describe our SCTP PROMELA model and properties that guide our analysis.

3.1 Overview

As we are primarily interested in denial-of-service attacks, and in order to avoid state-space explosion, we selectively model the SCTP connection establishment and teardown routines. This allows us to automatically and exhaustively explore, simulate, and verify the critical, security-relevant aspects of SCTP. Our model captures the following aspects of

SCTP per RFC 9260 [57]: internal peer states, packet verification using the itag and vtag, timers, TSNs, and handling for invalid, unexpected, and OOTB packets. We made only the abstractions listed in Section 3.4.

Although our model is fully faithful to the SCTP RFC [57], and is an executable program, it is not a network library and cannot be used in place of the existing Linux or FreeBSD implementations. This is because of the abstractions and simplifications mentioned above, and also because it does not implement API hooks for higher-level applications, nor syscalls to transmit over the Internet. It is simply a model of SCTP with which we can formally verify correctness properties.

3.2 Model Details

We describe our SCTP model in PROMELA, focusing on internal peer states, packet verification, invalid packet defense mechanisms, timeouts, and OOTB packet handling.

Mathematical Preliminaries. Linear Temporal Logic is a modal logic for reasoning about program executions. In LTL, we say a program P *models* a property ϕ, written $P \models \phi$, if ϕ holds over every execution of P. If ϕ holds over some but not all executions of P, then we write $P \not\models \phi$.

The LTL language is given by predicates (e.g., "Peer A is in Established" or "Peer B's cookie timer is inactive"); the temporal operators "next", "always", "eventually", and "until"; and the logical operators of negation, conjunction, and disjunction. An LTL model-checker is a tool that, given P and ϕ, can automatically check whether or not $P \models \phi$ [6]. We use the model-checker SPIN[7], whose language is PROMELA.

We use $\|$ to denote rendezvous composition, so, $S = P \| Q$ denotes that the program S equals the composition of P with Q. Specifically, matching *send* transitions of P and *receive* transitions of Q occur in lockstep, and vice versa. Note, in our model, we actually build a process called a "channel" to capture network delay, and we rendezvous-compose the channel with the two peers to build asynchronous communication (which is more realistic). The $\|$ operator is commutative and associative. For more details, refer to §2 of [64].

Internal Peer States. Our model consists of two peers (A and B) and a channel connecting them. That is, we study the system $S = \text{PEERA} \| \text{CHANNEL} \| \text{PEERB}$. Each peer is represented by an identical FSM, illustrated in Fig. 5. Transitions between states occur based on the receipt of user commands, or communication and message processing.

The channel connecting the two peers contains an internal single-message buffer in each direction (meaning it can hold two messages at once, one traveling from left to right and the other from right to left). It does not drop, corrupt, nor create messages, and cannot accept a new message in a given direction until the old one was delivered. In other words, it is lossless and FIFO, in that it guarantees every delivered message was sent and messages are delivered in order. The entire setup is illustrated in Figure 4.

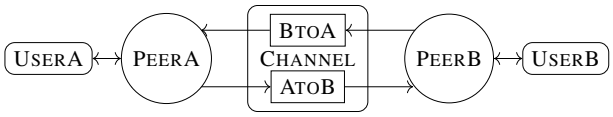

Figure 4: The system USERA $\|$ PEERA $\|$ CHANNEL $\|$ PEERB $\|$ USERB. CHANNEL contains a size-1 FIFO buffer in each direction (AtoB and BtoA, respectively). Arrows indicate communication direction. Composition between CHANNEL and peers is rendezvous (the buffers are inside CHANNEL).

Packet Verification and Invalid Packet Defenses. We model each SCTP message as consisting of a message chunk, a vtag, and an itag. Each of these components are modeled using enums, which in PROMELA are called `mtypes`. The message chunk denotes the meaning of the message, e.g., a message with an `INIT` chunk is called an *initiate message* and is used to initiate a connection establishment routine. The itag and vtag are used to verify the authenticity of the sender of the message, as described in Section 2.2. In our model there are three kinds of tags: expected (`E`), unexpected (`U`), or none (`N`). A tag is expected if (1) it is a non-zero itag on an `INIT` or `INIT_ACK` chunk, or (2) it is the other peer's vtag in the existing association. Otherwise, it is unexpected. The none type is reserved for packets that do not carry the given tag type – e.g., only `INIT` and `INIT_ACK` chunks carry an itag, so in the other types of messages, the itag is `N`. The BNF grammar for messages in our model is given in Figure 6. We also support an option where the *msg* can be extended with a TSN.

Upon receiving a message, our model checks that the tags are set as expected, depending on the message and state. If a message has an unexpected tag then the model employs the defenses specified in the RFC, e.g., silently discarding the message or responding with an `ABORT`. These defenses can be configured with or without the CVE patch from RFC 9260.

Active and Passive Connection Routines. Our SCTP model implements active/passive establishment and teardown, as well as active/active teardown, but not active/active establishment (a.k.a. "`INIT` collision"), precisely as described in Section 2 and illustrated in Figures 1 and 2, with the caveat that the itag and vtag are abstracted (as described above). We also capture the TSN proposal and use throughout an association, although this feature can be turned off in our model to reduce the state-space for more efficient verification.

Out-of-the-Blue and Unexpected Packets. Our model faithfully captures OOTB logic described in §8.4 of RFCs 4960 and 9260, with only the exceptions given in Section 3.4.

[6] LTL model-checking is decidable, and reduces to checking Büchi Automata intersection emptiness, which is PSPACE-complete.

[7] version 6.5.2

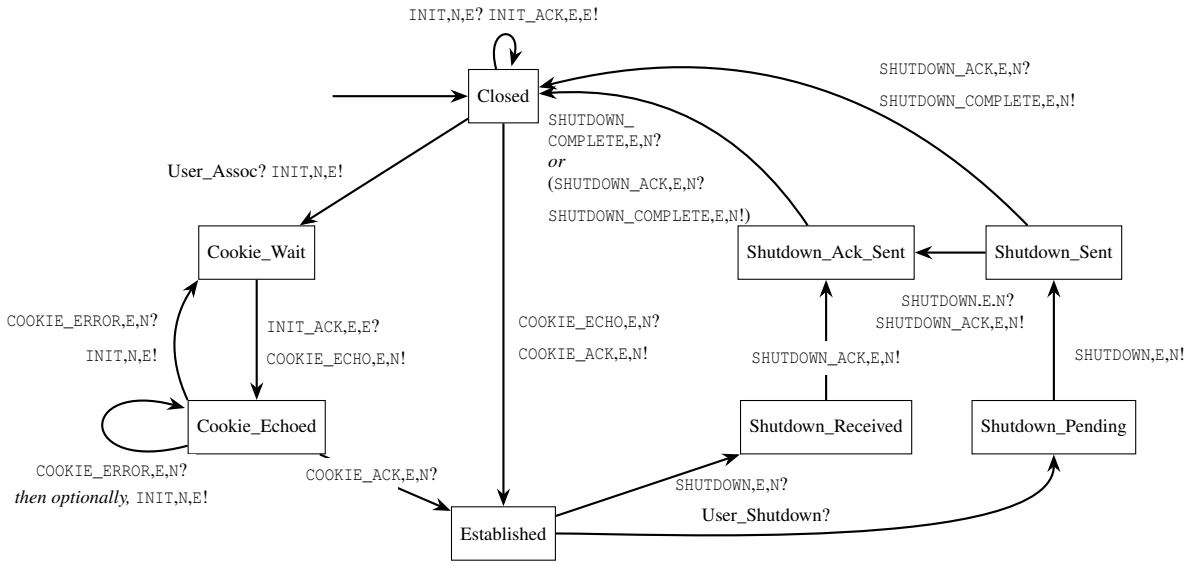

Figure 5: SCTP Finite State Machine. $x,v,i?$ (or $x,v,i!$) denotes receive (or send) chunk x with vtag v and itag i. Events in multi-event transitions occur in the order they are listed. Logic for OOTB packets, ABORT messages or User_Abort commands, unexpected user commands, and data exchange are ommitted but faithfully implemented in the model and described in this paper.

$$
\begin{aligned}
msg &::= \texttt{INIT},\texttt{N},ex \mid \texttt{INIT\_ACK},ex,ex \mid ach,ex,\texttt{N} \\
ach &::= \texttt{ABORT} \mid \texttt{SHUTDOWN} \mid \texttt{SHUTDOWN\_COMPLETE} \\
&\quad \mid \texttt{COOKIE\_ECHO} \mid \texttt{COOKIE\_ACK} \mid \texttt{SHUTDOWN\_ACK} \\
&\quad \mid \texttt{COOKIE\_ERROR} \mid \texttt{DATA} \mid \texttt{DATA\_ACK} \\
ex &::= \texttt{E} \mid \texttt{U}
\end{aligned}
$$

Figure 6: BNF grammar for messages in our model.

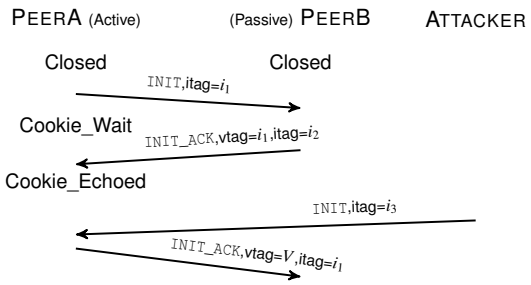

Figure 7: Ambiguous scenario. What value should V take? See Section 3.3.

3.3 Ambiguity in the RFC

We found one ambiguity in the SCTP RFCs, in §5.2.1, during the description of how a peer should react upon receiving an unexpected INIT chunk:

> *Upon receipt of an INIT chunk in the Cookie_Echoed state, an endpoint MUST respond with an INIT_ACK chunk using the same parameters it sent in its original INIT chunk (including its Initiate Tag, unchanged), provided that no new address has been added to the forming association.*

Consider two peers - A and B - initially both in Closed, in addition to some attacker who can spoof the port and IP of B. Suppose these machines engage in the sequence of events illustrated in Figure 7. At the end of the sequence, what value should the vtag V take?

To understand all possible interpretations, we ran the text through the AllenNLP Coreference Resolution tool [23], a state-of-the-art NLP model trained to detect which words in a sentence refer to the same entity, producing the following output, where entities are given ids and colored for readability; and each reference R to an entity with id I is highlighted I R.

> Upon receipt of an 1 INIT chunk in the Cookie_Echoed state, 0 an endpoint MUST respond with an INIT_ACK chunk using the same parameters 0 it sent in 0 its original 1 INIT chunk (including 0 its Initiate Tag, unchanged), provided that no new address has been added to the forming association.

The model predicts that the occurrences of *it* and *its* all refer to the same entity as *an endpoint*, which is clearly the responding endpoint, i.e., Peer A. What if a reader interprets "the same parameters" to include the vtag? Then the model would predict that the vtag of the INIT_ACK should come from the INIT that entity 0 sent, implying V should take the itag of the message, i.e. $V = i_1$. The fact that this is wrong only becomes clear if you fully understand how itags and vtags are used in both directions. To make the text unambiguous, we suggest adding the following sentence:

> The verification tag used in the packet containing the INIT_ACK chunk MUST be the initiate tag of the newly received INIT chunk.

The coreference resolution model predicts that "the newly received INIT chunk" is the same entity as the INIT chunk in "Upon receipt of an INIT chunk in the Cookie_Echoed state", so the text is unambiguous.

3.4 Abstractions and Limitations

Our model is fully faithful to the SCTP RFC, modulo the following abstractions and limitations.

- **Unicast peers.** In the RFC, OOTB messages from non-unicast peers are discarded; we model all peers as unicast.
- **No crashes or restarts.** In our model, peers never crash or restart. Thus we also ommit crash detection (including HEARTBEAT and HEARTBEAT_ACK chunks).
- **Tags are abstracted.** We do not model tie-tags, which are used when reconnecting a peer to an existing association after a restart. In the RFC, itags and vtags are integer-valued and chosen randomly. But we model tags as the "expected" value, an "unexpected" value, or "none", since this level of detail is all that matters for our properties. A side-effect is that we cannot study INIT collision. INIT collision is not included in the State Association Diagram in RFC 9260 §4, nor in the various association flows throughout the RFC document, leading us to believe it is not a protocol feature but rather an edge-case the protocol is designed to withstand.
- **Perfect channel.** We do not model packet loss, reordering, nor corruption, nor how SCTP deals with these scenarios.
- **Peers do not exchange data while in Established.** Because we focus on denial-of-service attacks, modeling data exchange while in Established is unnecessary; rather, we focused on the connection and disconnection of peers. We did model data transmission outside of Established, in case it caused edge-case behaviors during teardown.
- **Packets only ever contain one chunk.** Since we also do not model (or write properties about) fragmentation, bundling, or reassembly, we can simulate multi-chunk transmissions by sending consecutive single-chunk messages.
- **Simplified packet structure.** We choose not to model packet structure details relevant to only DATA packets, e.g.: stream sequence number, payload protocol identifier, and variable length. We also do not model ICMP messages.

3.5 Correctness Properties

Next we transcribe ten logical properties we believe SCTP should satisfy. Note, we do not intend to create a *complete* set of properties that captures all behaviors of SCTP. Rather, we design our properties to capture the security-relevant behavior of SCTP. Each property is implemented in PROMELA using LTL. We justify each using the RFCs [54,57] and our intuition about the security SCTP should provide.

ϕ_1**: A peer in Closed either stays still or transitions to Established or Cookie_Wait.** This is based on the routine described in §5.1, as well as the Association State Diagram in §4. If a closed peer could transition to any state other than Established or Cookie_Wait, it could de-synchronize with the other peer, breaking the four-way handshake and potentially leading to a deadlock, livelock, or other problem.

ϕ_2**: One of the following always eventually happens: the peers are both in Closed, the peers are both in Established, or one of the peers changes state.** The property we want to capture here, "no half-open connections", is stated in §1.5.1, was verified in the related work by Saini and Fehnker [52], and was studied for TCP in two prior works [46, 64]. But we have to formalize it subtly, because in the case of an in-transit ABORT, it is possible for one peer to temporarily be in Established while the other is in Closed; so we write it as a liveness property, saying half-open states eventually end.

ϕ_3**: If a peer transitions out of Shutdown_Ack_Sent then it must transition into Closed.** We derived this from the Association State Diagram in §4. Every transition out of Shutdown_Ack_Sent described in the RFC ends up in either Closed or Shutdown_Ack_Sent. If this property fails, it would imply a flaw in the graceful teardown routine, and could cause a deadlock, livelock, or other problem.

ϕ_4**: If a peer is in Cookie_Echoed then its cookie timer is actively ticking.** Per §5.1 C), the peer starts the cookie timer upon entering Cookie_Echoed. Per §4 step 3), when the timer expires it is reset, up to a fixed number of times, at which point the peer returns to Closed. If the property fails, then the active peer in an establishment could get stuck in Cookie_Echoed forever, opening a new opportunity for DoS.

ϕ_5**: The peers are never both in Shutdown_Received.** This property follows from inspection of the Association State Diagram in §4. From a security perspective, if both peers were in Shutdown_Received, this would indicate that neither initiated the shutdown (yet both are shutting down); the only logical explanation for which is some kind of DoS.

ϕ_6**: If a peer transitions out of Shutdown_Received then it must transition into either Shutdown_Ack_Sent or Closed.** The transition to Shutdown_Ack_Sent is shown in the Association State Diagram in §4. The transition to Closed can occur upon receiving either a User_Abort from the user or an ABORT from the other peer. No other transitions out of Shutdown_Received are given in the RFC. If this property fails, it could de-synchronize the teardown handshake, potentially leading to an unsafe behavior. For example, if a peer transitioned from Shutdown_Received to Established, it would end up in a half-open connection.

ϕ_7**: If Peer A is in Cookie_Echoed then B must not be in Shutdown_Received.** We derived this from the Association Diagram in §4, which shows A must receive an INIT_ACK while in the Cookie_Wait and then send a COOKIE_ECHO in order to transition into Cookie_Echoed. B must have been in Closed to send an INIT_ACK in the first place, hence B

cannot be in Shutdown_Received. This property relates to the synchronization between the peers: if one is establishing a connection while the other is tearing down, then they are de-synchronized, and the protocol has failed.

ϕ_8**: Suppose that in the last time-step, Peer A was in Closed and Peer B was in Established. Suppose neither user issued a User_Abort, and neither peer had a timer time out. Then if Peer A changed state, it must have changed to either Established, or the implicit, intermediary state in Cookie_Wait in which it received `INIT_ACK` but did not yet transmit `COOKIE_ECHO`.** The transitions from Closed to Established and the described intermediary state are implicit in the Association State Diagram in §4. The timer caveat is described in §4 step 2, and the aborting caveat is in §9.1. If the property fails, the four-way handshake ended, yet was not completed successfully, did not time out, and was not aborted, so somehow, the protocol failed.

ϕ_9**: The same as ϕ_8 but the roles are reversed.** The property is: *Suppose that in the last time-step, Peer B was in Closed and Peer A was in Established...*

ϕ_{10}**: Once connection termination initiates, both peers eventually reach Closed**. This follows from the description of connection termination in §9. Once connection termination is initiated, there is no way to recover the association.

For the On-Path attacker model, ϕ_8 and ϕ_9 are symmetric. For the other attacker models, the properties are distinct, because the attacker model's network topology is asymmetric.

3.6 Validating Our Model

Our model allows us to execute and reason about any component of the peer logic in isolation, or two interacting peers. To verify our model, we extracted the properties listed above from the SCTP RFCs, and then used the model-checker to prove that our model satisfies all of the properties. We interactively guided SPIN to drive the model through various connection flows (which we compared to the RFC text), and we manually compared our logic for handling OOTB packets to the corresponding C code in Linux and FreeBSD. Finally, we used SPIN to prove there were no deadlocks or livelocks (liveness cycles) and all the peer states are reachable.

4 SCTP Attack Synthesis

In this section we provide details on attack synthesis and KORG, the tool we used. Next, we describe four attacker models we defined and used for our analysis. Finally, we present the changes we had to make to KORG to handle our SCTP model, and the four attacker models we considered.

4.1 Attack Synthesis

LTL program synthesis is the problem of, given an LTL specification ϕ, automatically deriving a compliant program P (for which $P \models \phi$). *LTL attack synthesis* is fundamentally different (logically dual), and cannot be solved using program synthesis alone. In attack synthesis, the problem is flipped: given a program S and property ϕ, where S is already compliant ($S \models \phi$), if $S = P \parallel Q$ consists of an *invariant component* P (that the attacker cannot change) and a *variant component* Q (that the attacker can change), we ask whether there exists some modification A such that, if we replace Q with A, the new system $S' = P \parallel A$ is non-compliant ($S' \not\models \phi$). In other words, we study a system that behaves correctly, and ask if we can change some constrained aspect so that it behaves incorrectly. If so, we call this modification A an "attack".

There are multiple kinds of attacks one might try to synthesize, depending on the nature of the protocol and the attacker goal. We use use KORG [64], which leverages SPIN [28] to synthesize attacks against arbitrary LTL properties of transport protocols. KORG was previously successfully applied to TCP and DCCP [46,64], and to the best of our knowledge, it is the only open-source attack synthesis tool that can synthesize terminating fixed-vocabulary communication attacks against arbitrary LTL properties.[8] KORG is proven to be sound (it has no false-positives) and complete (if attacks of the kind KORG looks for exist, given enough resources, KORG will find one). Meanwhile, SPIN has existed for 35 years; has been applied to dozens of real systems including the Mars rover [29], PathStar access server [30], and ISO/IEEE P11073-20601 medical communication protocol [24]; spawned a dedicated formal methods conference, currently in its 30th year[9]; and earned the the 2002 ACM Software System Award.

On the other hand, PROVERIF [13] and TAMARIN [42] are designed to verify, or synthesize attacks against, secrecy, authentication, privacy, and equivalence properties of cryptographic protocols using the Dolev-Yao attacker model [18]. Although TAMARIN admits a small language of guarded first-order trace properties, in practice it does not scale to complicated properties (e.g., ϕ_8) of complex models (e.g., ours) [10], the language is not as well-suited as LTL for verifying state-based properties of transport protocols, and it does not support arbitrary attacker models (such as ours, which differ from Dolev-Yao – e.g., because in Dolev-Yao the attacker can spawn infinitely many associations at once, but in our attacker models, it is limited to one at a time). Thus, PROVERIF and TAMARIN [42] are better than KORG for studying cryptographic protocols such as TLS [11] or Signal [34] under Dolev-Yao, but KORG is better for synthesizing communication attacks against non-cryptographic transport protocols such as SCTP under custom attacker models.

KORG requires four inputs: an invariant component P (e.g., the SCTP model) and variant component Q (which in our case is part of the attacker model), both in PROMELA; an LTL

[8] We discuss another approach [40] based on reactive controller synthesis in Sec. 6, but it is not directly comparable as it synthesizes a narrow category of attacks that are guaranteed to succeed, which do not always exist.

[9] https://spin-web.github.io/SPIN2023/

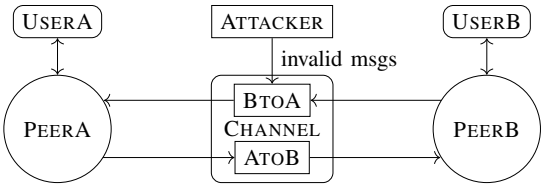

Figure 8: Off-Path Attacker Model: S = Attacker ‖ UserA ‖ PeerA ‖ Channel ‖ PeerB ‖ UserB. The attacker can transmit messages into the BtoA buffer, but cannot receive messages, nor block messages in-transit. The attacker can send only chunks having an invalid itag and vtag (as it is not privy to the association).

correctness property ϕ, such that the composite system consisting of both P and Q satisfies ϕ ($P \parallel Q \models \phi$); and a YAML file encoding the grammar (I/O) of Q (which become the I/O of the attacker). KORG generates an model with these inputs in which Q is replaced with a process called a *daisy*, that can nondeterministically send or receive messages specified in the grammar. Next, it modifies ϕ to have a precondition saying the daisy terminates, and then asks SPIN to verify or disprove the modified property for the modified model. Either KORG reports no attacks exist, or SPIN outputs a counterexample execution, which KORG parses into an attack A. For more refer to [64]. The inputs to KORG needed to reproduce each of our experiments are documented in the Appendix in Table 4.

KORG is limited by the level of detail in the model, the definition of "attack" used by KORG [64], and the attacker models and properties considered. Thus there could exist other attacks beyond those KORG synthesizes, which violate other properties or work in other attacker models; or attacks other than the type that KORG can find (e.g., statistical ones); or attacks that cannot be found without a more detailed model. These limitations are inherent to all attack synthesis techniques.

4.2 Attacker Models

We use the term *attacker model* to mean a formal description of the placement and capabilities of the attacker and protocol peer(s) on the network. We create four attacker models: Off-Path, Evil-Server, Replay, and On-Path. They are general-purpose and applicable to any transport protocol, and we contribute them to KORG.

Off-Path. In this model, an attacker who does not know either vtag communicates with one peer in order to disrupt the association formed by the two peers that want to communicate. The vtag mechanism in SCTP was designed to defend against such an attacker. See Figure 8.

Evil-Server. In this model, one of the peers behaves maliciously. For example, the attacker takes the form of a finite sequence of malicious instructions inserted before the code of Peer B, after which B behaves like normal. See Figure 9.

Replay. In this model, the attacker can replay captured packets without modification. See Figure 10.

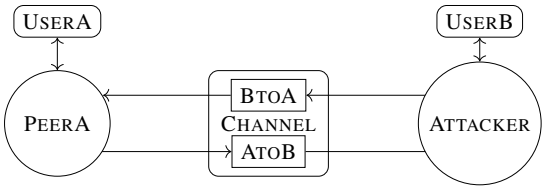

Figure 9: Evil-Server Attacker Model: S = UserA ‖ PeerA ‖ Channel ‖ Attacker ‖ UserB. Peer B is prefixed with an attacker, whose code consists of a finite, terminating sequence of communication operations.

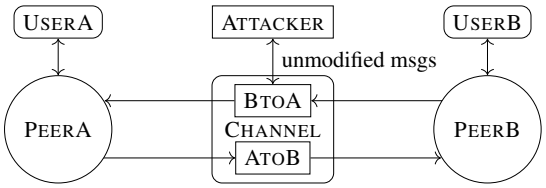

Figure 10: Replay Attacker Model: S = Attacker ‖ UserA ‖ PeerA ‖ Channel ‖ PeerB ‖ UserB. The attacker can capture and re-transmit messages in BtoA, but cannot edit captured messages, nor block those in transit.

On-Path. . In this attacker model, the attacker controls the channel connecting the two peers, and can drop or insert valid messages at-will. SCTP was not designed to provide security against such an attacker and we study this attacker model only to understand what the "worst case scenario" for SCTP looks like. See Figure 11.

4.3 Changes to KORG

We improved KORG to support our SCTP analysis in four ways. (1) Since KORG was originally hard-coded for enum-style packets, we extended KORG to support arbitrary finite packet types. This was needed to support our SCTP model (Figure 6). (2) We modified KORG to report any attacks it finds even if it fails to exhaust the search-space. Previously, it would report an error and discard any results if the space was not exhausted. (3) To save time, we disabled the preliminary step

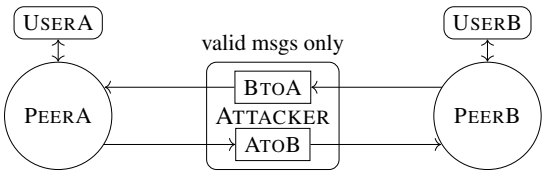

Figure 11: On-Path Attacker Model: S = UserA ‖ PeerA ‖ Attacker ‖ PeerB ‖ UserB. The attacker is allowed to perform a finite sequence of send/receive actions, in which it only sends valid messages (but can receive anything). Once this sequence terminates, it behaves like an honest channel.

where KORG verifies that the property holds in the absence of an attacker, instead manually performing this step in SPIN. (4) We extended KORG to support support replay attackers.

A replay attacker is one capable of capturing and replaying messages. Although the replay attacker model reasons about packets received, the attacks this model produces form a subset of those produced by the *On-Path* attacker; thus, soundness and completeness follow from the proofs in the KORG paper. Our replay attacker synthesis implementation supports packet capture and replay over the same or different channels. In the latter case, the attacker can capture a message from one channel and replay it into another. It also supports packet storage in a memory buffer with configurable size, though, the verification complexity increases exponentially with the memory bound. Finally, to support the state change that happens when a new vtag is chosen, we added a feature where a special message can be configured to flush the storage.

We also contribute our SCTP model and four attacker models in a format amenable to KORG. We document the attacker models in Section 4.2. Excluding the models, our modifications required changing 80 lines of preexisting code and adding 213 lines of new functionality in KORG, in addition to 102 lines of shell-script to automate our experiments. All modifications are available with the paper artifacts.

5 Experimental Results

We next present our experimental results. Our SCTP model satisfies all ten properties in the absence of an attacker. To examine whether these properties still hold when an attacker is present, we synthesize attacks using the three attacker models. Then, we enable the CVE patch described in RFC 9260 and repeat our analysis, in order to check whether the patch resolves the vulnerability, and/or introduces any new attack vectors. Finally, we show how a new attack is enabled if the ambiguous text identified in Section 3.3 is misinterpreted. Analysis runtimes are related in Section 8.4 in the appendix.

5.1 Experimental Methodology

Each time we run KORG, we ask it to synthesize ≤ 10 attacks. In our experience, after the first ten, subsequent attacks tend to be repetitive, differing only by actions that do not impact the attack outcome. We configure KORG with a default search depth of 600,000, and a maximum depth of 2,400,000. In our experience, these parameters balance fast-performance on smaller properties with the ability to also attack more complex ones, without needing to run on a cluster. We make certain assumptions or optimizations in the different attacker models.

• *Off-Path:* We assume the Off-Path attacker knows the port and IP of a peer, since otherwise, all its (spoofed) messages will be immediately discarded.[10] To reduce the search-space, we assume the attacker does not send `DATA` or `COOKIE_ERROR` chunks, which cannot change the receiving peer's state. We further reduce the space by first synthesizing attacks against the establishment routine, where the attacker could only send messages that are used during establishment; and then doing the same for teardown. Our search is complete despite this split because the FSM is inherently Markovian and our properties do not look back more than one state in the past. In our open-source artifacts, we provide code illustrating how this optimization can be repeated for any transport protocol.

• *Evil-Server:* We assume the Evil-Server attacker only sends valid messages, since it knows the current vtags. To reduce the search-space, we assume it does not send `DATA`.

• *Replay:* We configure the replay attacker to have a memory size of two — we more memory causes state-space explosion and makes exhaustive verification infeasible. We also configure it to discard all messages in memory upon receiving an `INIT` chunk, as this allows us to correctly model the vtag change between multiple connection and teardown cycles.

• *On-Path:* We perform the same optimizations as in the Off-Path attacker model. And like in the Evil-Server attacker model, we assume the attacker only sends valid messages.

Our experiments are easily reproducible using the paper artifacts. We summarize the inputs given to KORG for each experiment in Table 4 in the Appendix.

[10] The ports and IP of a peer might not change between associations [56].

```
BTOA!COOKIE_ACK,U,N;        (repeat twice more)
BTOA!COOKIE_ECHO,U,N;
BTOA!COOKIE_ACK,U,N;        (repeat 6 more times)
BTOA!COOKIE_ECHO,U,N;
BTOA!INIT,N,U;                    /* attack */
```

Figure 12: Automatically synthesized CVE attack in the Off-Path attacker model. BTOA is the channel from the attacker to the peer being attacked. Only the final line matters.

5.2 Attacks

We generate at least one attack in each attacker model, all of which we summarize in Table 2. We discuss results for each attacker model in detail below.

Off-Path. KORG found a variant of the attack reported in CVE-2021-3772, given in Figure 12. The variant differs only from the CVE in that it begins by transmitting some OOTB messages that are discarded and have no impact on the outcome. It ends with the transmission of an `INIT` with an unexpected (zero) itag, which is the CVE attack.

Evil-Server. KORG synthesizes four attacks. The first attack models a scenario in which the Shutdown Timer is configured to a very large value, and thus the attacker can cause a peer in active teardown to (essentially) deadlock by never

Attacker Model	Property	Synthesized Attacks
Off-Path	ϕ_9	A single variant of the CVE attack.
Evil-Server	ϕ_1	One attack where the attacker guides A through passive establishment. Then when A attempts active teardown, if its Shutdown Timer never fires, it deadlocks.
	ϕ_6	One attack where the attacker guides A to Shutdown_Received, then sends it an unexpected `COOKIE_ECHO`, causing it to go back to Established.
	ϕ_8	One attack where the attacker guides A through most of active establishment before aborting the connection. When the attack terminates, B receives the en-route `COOKIE_ECHO` and completes passive establishment, creating a half-open connection.
	ϕ_9	One attack where the attacker guides A through passive establishment then terminates. If B then attempts active establishment, the property fails, since the peers are de-synchronized.
Replay	ϕ_2	One attack where the attacker sends an `ABORT` before the peers establish a TSN for the association.
On-Path	ϕ_5	Four attacks where the attacker manipulates both peers into Shutdown_Received.
	ϕ_8	Two attacks where the attacker spoofs A to guide B through passive establishment.
	ϕ_9	Two attacks where the attacker spoofs B to guide A through passive establishment.

Table 2: Attacks found.

responding to its `SHUTDOWN` message. In the second attack, the attacker exploits the unexpected packet logic in §5.2.4 to guide a peer out of passive teardown and back into Established. The third and fourth attacks are similar and involve guiding one peer through establishment to de-synchronize it with the other, leading to a half-open connection.

Replay. KORG synthesizes one attack where the attacker captures and replays an `ABORT` message sent by a peer before both peers establish a new TSN. The attacker can keep replaying the `ABORT` message indefinitely, preventing the peers from establishing a connection. No other attacks were found. This is expected, as the OOTB logic and TSNs should prevent a replay attacker from injecting old packets.

On-Path. KORG synthesizes four similar attacks where the attacker guides both peers into an association, and then spoofs each peer, sending a `SHUTDOWN` to the other. In general, an On-Path attacker is so powerful that we expect it can manipulate either peer into any state it pleases, as it totally controls the network, so this is unsurprising. KORG synthesizes two more attacks, one for each of the half-open properties, both similar to the last attack reported with the Evil-Server attacker model.

Note, the reason we do not rediscover the CVE attack in the Evil-Server or On-Path attacker model is that we restrict the attacker in both to only send valid messages, whereas the CVE attack requires an invalid `INIT`. We put this restriction in place in our model to avoid state-space explosion. In general, every attack that is possible in the Off-Path attacker model is also possible in the Evil-Server and On-Path ones.

5.3 Patch Verification

Next, we re-run our analysis with the CVE patch enabled. In the Off-Path attacker model, KORG terminates without finding any attacks. Since KORG found the vulnerability in the Off-Path attacker model when the patch was disabled, and reports no attacks in that same attacker model when the patch is enabled, and as KORG is complete, this suffices to prove that the patch resolves the vulnerability. In the other attacker models, we find the exact same attacks as those reported in Table 2, and nothing more. This proves the patch does not introduce any new attack vectors against our properties.

5.4 Ambiguity Analysis

We next configure our model with the incorrect interpretation of the ambiguous text and run it interactively in SPIN. We find that the incorrect interpretation could enable a denial-of-service in the form of a half-open connection, which we illustrate in Figure 13. We consulted with the lead SCTP RFC author who confirmed that the misinterpretation we describe could enable such an attack. The attack is not possible if the text is interpreted correctly. Out of concern that a real implementation might have misinterpreted the RFC document, we manually analyzed the source for both the Linux and FreeBSD implementations, and tested both implementations with PACKETDRILL, finding that neither made this mistake.

6 Related Work

There are many automated attack discovery tools, each crafted to a particular variety of bug or mechanism of attack, e.g., SNAKE [32] (which fuzzes network protocols), TCPWN [31] (which finds throughput attacks against TCP congestion control implementations), TAMARIN [42] and PROVERIF [13] (which find attacks against secrecy in cryptographic protocols), KORG [64] (which finds communication attacks against network protocols), and so on. Some of these tools (e.g., KORG) are general purpose, designed to attack any correct-

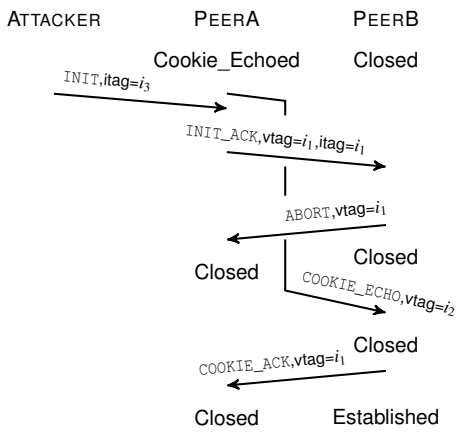

Figure 13: Message sequence chart showing the vtag-disclosure vulnerability enabled by misinterpretation of the ambiguous RFC text reported in Section 3.3. Note the strict timing requirements necessary for a successful attack.

ness property, while others (e.g., TAMARIN or PROVERIF) are designed to target specific types of properties such as secrecy and trace-equivalence. One work, which studied TCP and ABP, suggested reactive controller synthesis (RCS) as an alternative to KORG's approach [40]. KORG generates attacks that sometimes succeed, depending on choices made by the peers, whereas the RCS method only outputs attacks that always succeed; but such attacks do not always exist. Another approach, which Fiterau-Brostean et. al. [20] successfully applied to various SSH [67] and DTLS [50] implementations, describes incorrect behaviors using automata (rather than properties). This specification style makes sense when generic bug patterns are known ahead of time.

Formal methods such as theorem proving, model checking, property-based testing, and attack synthesis for protocols have been applied to TLS [17] and accountable proxying over it [12], QUIC [41], Bluetooth [66], 5G [7] and its key-establishment stack [43], TCP congestion control [6] and the combination of Karn's Algorithm and the RTO computation upon which it relies [63], the TCP establishment routine [44, 46, 64], and contactless EMV payments [8, 47], to name a few. Compared to many of these systems, such as TCP which has been studied for over 30 years [9, 25, 27, 31, 32, 46, 53, 64], much less is known about the security of SCTP, particularly from an FM perspective.

Of the prior works that applied formal methods to the security of SCTP, only the Uppaal analysis by Saini and Fehnker [52] used a technology (model-checking) that can verify arbitrary properties. They reported two properties in their paper; the first is similar to our ϕ_2. The second says an adversary only capable of sending INIT packets cannot cause a victim peer to change state. This property is trivial for us because we use an FSM model where the peer states are precisely the model states. And in our model, the only transition out of Closed that happens upon receiving an INIT is a self-loop that sends an INIT_ACK and returns to Closed. In contrast, in Saini and Fehnker's model the peer state is a variable in memory, while the model states are totally different (e.g., LC1, LC2). Thus, the property merits verification in their model but not ours. Saini and Fehnker's work is the only one we are aware of that studied SCTP in the context of an attacker using formal methods. But their attacker was only capable of sending INIT messages, in contrast to our attacker models which are much more sophisticated, and their attacker could not spoof the port and IP of a peer. Hence, they could not model (and so did not find) the CVE attack.

Another line of inquiry aims to model the performance of SCTP, e.g., using numerical analyses and simulations [16]. For example, Fu and Atiquzzaman built an analytical model of SCTP congestion control, including *multihoming*, an SCTP feature not available in TCP. They compared their model to simulations and found it to be accurate in estimating steady-state throughput of multihomed paths [22]. Such models are also used to evaluate new features, e.g., as in [68].

7 Conclusion

In this work we formally modeled SCTP and specified ten novel LTL correctness properties based on a close reading of the RFCs. We proved that in the absence of an attacker, the protocol satisfies all ten properties. We used KORG to synthesize attacks against our model for four novel attacker models, Off-Path, Evil-Server, Replay, and On-Path, and for two configurations of the SCTP model – one without the RFC 9260 patch and another with it. This required improvements to KORG, which we open-sourced with the paper artifacts. Without the patch, we found the CVE-2021-3772 attack in the Off-Path attacker model; a variety of Evil-Server and On-Path attacks; and one Replay attack. Then we repeated our analysis with the patch, and found that it eliminated the CVE vulnerability but did not eliminate any other attacks in other attacker models, nor introduce new vulnerabilities.

We also explored extending KORG to not just discover vulnerabilities, but synthesize patches too. We found the task infeasible as the search space for edits is enormous, and each edit requires re-verifying. And since PROMELA does composition over FIFO channels, reasoning about the composite Kripke Structure and tying it back to the PROMELA encoding proved very challenging. Though we failed to synthesize patches in this work, we believe patch synthesis may be plausible in a more automata-theoretic context.

Our attacks highlight the need to explicitly handle unexpected but valid packets and set reasonable timer values. We reported an ambiguity in RFC 9260, one interpretation of which could lead to a vulnerability. We analyzed the Linux and FreeBSD SCTP implementations using PACKETDRILL and found both correctly interpreted the ambiguous text. We concluded with a recommendation for how the text could be

made unambiguous in an erratum or future RFC.

References

[1] packetdrill. https://github.com/nplab/packetdrill/tree/master. Commit 7f3daabd7feed2b18b958e870f973fec92879d98, accessed 31 July 2023.

[2] SCTP. https://github.com/torvalds/linux/tree/master/net/sctp. Accessed 15 March 2023.

[3] User Datagram Protocol. RFC 768, Aug. 1980.

[4] Sctp. https://man.freebsd.org/cgi/man.cgi?query=sctp&sektion=4&manpath=FreeBSD+7.0-RELEASE, 2006. Accessed 1 May 2023.

[5] Data communication. https://webrtcforthecurious.com/docs/07-data-communication/, November 2022. Accessed 31 July 2023.

[6] ARUN, V., ARASHLOO, M. T., SAEED, A., ALIZADEH, M., AND BALAKRISHNAN, H. Toward formally verifying congestion control behavior. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (2021), pp. 1–16.

[7] BASIN, D., DREIER, J., HIRSCHI, L., RADOMIROVIC, S., SASSE, R., AND STETTLER, V. A formal analysis of 5g authentication. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security* (2018), pp. 1383–1396.

[8] BASIN, D., SASSE, R., AND TORO-POZO, J. The emv standard: Break, fix, verify. In *2021 IEEE Symposium on Security and Privacy (SP)* (2021), IEEE, pp. 1766–1781.

[9] BELLOVIN, S. M. Security problems in the tcp/ip protocol suite. *ACM SIGCOMM Computer Communication Review 19*, 2 (1989), 32–48.

[10] BEN HENDA, N. Generic and efficient attacker models in spin. In *Proceedings of the 2014 international SPIN symposium on model checking of software* (2014), pp. 77–86.

[11] BHARGAVAN, K., BLANCHET, B., AND KOBEISSI, N. Verified models and reference implementations for the tls 1.3 standard candidate. In *2017 IEEE Symposium on Security and Privacy (SP)* (2017), IEEE, pp. 483–502.

[12] BHARGAVAN, K., BOUREANU, I., DELIGNAT-LAVAUD, A., FOUQUE, P.-A., AND ONETE, C. A formal treatment of accountable proxying over tls. In *2018 IEEE Symposium on Security and Privacy (SP)* (2018), IEEE, pp. 799–816.

[13] BLANCHET, B., ET AL. Modeling and verifying security protocols with the applied pi calculus and proverif. *Foundations and Trends® in Privacy and Security 1*, 1-2 (2016), 1–135.

[14] CARDWELL, N., CHENG, Y., BRAKMO, L., MATHIS, M., RAGHAVAN, B., DUKKIPATI, N., CHU, H.-K. J., TERZIS, A., AND HERBERT, T. PACKETDRILL: Scriptable network stack testing, from sockets to packets. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)* (2013), pp. 213–218.

[15] CARO, A. L., POON, K., TÜXEN, M., STEWART, R. R., AND ARIAS-RODRIGUEZ, I. Stream Control Transmission Protocol (SCTP) Specification Errata and Issues. RFC 4460, Apr. 2006.

[16] CHUKARIN, A., AND PERSHAKOV, N. Performance evaluation of the stream control transmission protocol. In *MELECON 2006-2006 IEEE Mediterranean Electrotechnical Conference* (2006), IEEE, pp. 781–784.

[17] CREMERS, C., HORVAT, M., HOYLAND, J., SCOTT, S., AND VAN DER MERWE, T. A comprehensive symbolic analysis of tls 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), pp. 1773–1788.

[18] DOLEV, D., AND YAO, A. On the security of public key protocols. *IEEE Transactions on information theory 29*, 2 (1983), 198–208.

[19] EDDY, W. Transmission Control Protocol (TCP). RFC 9293, Aug. 2022.

[20] FITERAU-BROSTEAN, P., JONSSON, B., SAGONAS, K., AND TÅQUIST, F. Automata-based automated detection of state machine bugs in protocol implementations. In *NDSS* (2022).

[21] FLOYD, S., HANDLEY, M. J., AND KOHLER, E. Datagram Congestion Control Protocol (DCCP). RFC 4340, Mar. 2006.

[22] FU, S., AND ATIQUZZAMAN, M. Performance modeling of SCTP multihoming. In *GLOBECOM'05. IEEE Global Telecommunications Conference, 2005.* (2005), vol. 2, IEEE, pp. 6–pp.

[23] GARDNER, M., GRUS, J., NEUMANN, M., TAFJORD, O., DASIGI, P., LIU, N., PETERS, M., SCHMITZ, M., AND ZETTLEMOYER, L. Allennlp: A deep semantic natural language processing platform. *arXiv preprint arXiv:1803.07640* (2018).

[24] GOGA, N., COSTACHE, S., AND MOLDOVEANU, F. A formal analysis of iso/ieee p11073-20601 standard of medical device communication. In *2009 3rd Annual IEEE Systems Conference* (2009), IEEE, pp. 163–166.

[25] GONT, F. ICMP attacks against TCP. https://datatracker.ietf.org/doc/rfc5927/, july 2022.

[26] GUO, C., AND TÜEXEN, M. Questions on rfc4960 abort init tag equal 0 in init msg and manditory info less than 20 bytes. https://mailarchive.ietf.org/arch/msg/tsvwg/nna1IVrRIKPBKOwmv5JC3X5UQSA/. Accessed 5 February 2024.

[27] HARRIS, B., AND HUNT, R. Tcp/ip security threats and attack methods. *Computer communications 22*, 10 (1999), 885–897.

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

[29] HOLZMANN, G. J. Redundant software (and hardware) ensured curiosity reached its destination and functioned as its designers intended. *Communications of the ACM* (2 2014).

[30] HOLZMANN, G. J., AND SMITH, M. H. Automating software feature verification. *Bell Labs Technical Journal 5*, 2 (2000), 72–87.

[31] JERO, S., HOQUE, M. E., CHOFFNES, D. R., MISLOVE, A., AND NITA-ROTARU, C. Automated attack discovery in tcp congestion control using a model-guided approach. In *NDSS* (2018).

[32] JERO, S., PACHECO, M. L., GOLDWASSER, D., AND NITA-ROTARU, C. Leveraging textual specifications for grammar-based fuzzing of network protocols. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2019), vol. 33, pp. 9478–9483.

[33] KHOT, I. A smaller, faster video calling library for our apps. https://engineering.fb.com/2020/12/21/video-engineering/rsys/, december 2020. Accessed 31 July 2023.

[34] KOBEISSI, N., BHARGAVAN, K., AND BLANCHET, B. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *2017 IEEE European symposium on security and privacy (EuroS&P)* (2017), IEEE, pp. 435–450.

[35] KUMAR, A., VON HIPPEL, M., MANOLIOS, P., AND NITA-ROTARU, C. Formal model-driven analysis of resilience of gossipsub to attacks from misbehaving peers. *arXiv preprint arXiv:2212.05197* (2022).

[36] LOHR, H., AND DKNAPPETTMSFT. What's new in the remote desktop WebRTC redirector service. https://learn.microsoft.com/en-us/azure/virtual-desktop/whats-new-webrtc, April 2023. Accessed 31 July 2023.

[37] LONG, X. SCTP enhancements for the verification tag. https://github.com/torvalds/linux/commit/32f8807a48ae55be0e76880cfe8607a18b5bb0df. Accessed 5 February 2024.

[38] LONG, X. https://github.com/torvalds/linux/commit/32f8807a48ae55be0e76880cfe8607a18b5bb0df, October 2021.

[39] MARTINS, M. G. M., ET AL. Modelagem e análise formal de algumas funcionalidades de um protocolo de transporte através das redes de petri. Accessed 2 August 2023 at https://docplayer.com.br/146114380-Modelagem-e-analise-formal-de-algumas-funcionalidades-de-um-protocolo-de-transporte-atraves-das-redes-de-petri.html.

[40] MATSUI, S., AND LAFORTUNE, S. Synthesis of winning attacks on communication protocols using supervisory control theory: two case studies. *Discrete Event Dynamic Systems* (2022), 1–38.

[41] MCMILLAN, K. L., AND ZUCK, L. D. Formal specification and testing of QUIC. In *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 227–240.

[42] MEIER, S., SCHMIDT, B., CREMERS, C., AND BASIN, D. The tamarin prover for the symbolic analysis of security protocols. In *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25* (2013), Springer, pp. 696–701.

[43] MILLER, R., BOUREANU, I., WESEMEYER, S., AND NEWTON, C. J. The 5g key-establishment stack: In-depth formal verification and experimentation. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security* (2022), pp. 237–251.

[44] MUSUVATHI, M., ENGLER, D. R., ET AL. Model checking large network protocol implementations. In *NSDI* (2004), vol. 4, pp. 12–12.

[45] OAKLEY, L., OPREA, A., AND TRIPAKIS, S. Adversarial robustness verification and attack synthesis in stochastic systems. In *2022 IEEE 35th Computer Security Foundations Symposium (CSF)* (2022), IEEE, pp. 380–395.

[46] PACHECO, M. L., VON HIPPEL, M., WEINTRAUB, B., GOLDWASSER, D., AND NITA-ROTARU, C. Automated attack synthesis by extracting finite state machines from protocol specification documents. In *2022 IEEE Symposium on Security and Privacy (SP)* (2022), IEEE, pp. 51–68.

[47] RADU, A.-I., CHOTHIA, T., NEWTON, C. J., BOUREANU, I., AND CHEN, L. Practical emv relay protection. In *2022 IEEE Symposium on Security and Privacy (SP)* (2022), IEEE, pp. 1737–1756.

[48] RED HAT, I. CVE-2021-3772 detail. https://nvd.nist.gov/vuln/detail/CVE-2021-3772. Accessed 15 March 2023.

[49] RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, Aug. 2018.

[50] RESCORLA, E., TSCHOFENIG, H., AND MODADUGU, N. The datagram transport layer security (DTLS) protocol version 1.3. https://www.rfc-editor.org/rfc/rfc9147, April 2022.

[51] RÜNGELER, I., AND TÜXEN, M. Sctp support in the inet framework and its analysis in the wireshark packet analyzer. In *Multihomed Communication with SCTP (Stream Control Transmission Protocol)*. CRC Press, 2012, pp. 175–202.

[52] SAINI, S., AND FEHNKER, A. Evaluating the stream control transmission protocol using Uppaal. *arXiv preprint arXiv:1703.06568* (2017).

[53] SCHUBA, C. L., KRSUL, I. V., KUHN, M. G., SPAFFORD, E. H., SUNDARAM, A., AND ZAMBONI, D. Analysis of a denial of service attack on tcp. In *Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No. 97CB36097)* (1997), IEEE, pp. 208–223.

[54] STEWART, R. Stream control transmission protocol. https://www.rfc-editor.org/rfc/rfc4960, September 2007. Accessed 23 February 2023.

[55] STEWART, R., TUEXEN, M., AND CAMARILLO, G. Security attacks found against the stream control transmission protocol (SCTP) and current countermeasures. https://datatracker.ietf.org/doc/html/rfc5062, September 2007.

[56] STEWART, R., TÜXEN, M., AND LEI, P. SCTP: What is it, and how to use it? In *Proceedings of BSDCan: The Technical BSD Conference* (2008).

[57] STEWART, R., TÜXEN, M., AND NIELSEN, K. Stream control transmission protocol. https://www.rfc-editor.org/rfc/rfc9260, June 2022. Accessed 15 March 2023.

[58] STEWART, R., XIE, Q., MORNEAULT, K., SHARP, C., SCHWARZBAUER, H., TAYLOR, T., RYTINA, I., KALLA, M., ZHANG, L., AND PAXSON, V. Stream control transmission protocol. https://www.rfc-editor.org/rfc/rfc2960, October 2000. Accessed 15 March 2023.

[59] VANIT-ANUNCHAI, S. Towards formal modelling and analysis of SCTP connection management. In *Proceedings of the Ninth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools* (2008).

[60] VANIT-ANUNCHAI, S. Validating SCTP simultaneous open procedure. In *Fundamentals of Software Engineering: 5th International Conference, FSEN 2013, Tehran, Iran, April 24-26, 2013, Revised Selected Papers 5* (2013), Springer, pp. 233–249.

[61] VARDI, M. Y., AND WOLPER, P. An automata-theoretic approach to automatic program verification. In *1st Symposium in Logic in Computer Science (LICS)* (1986), IEEE Computer Society.

[62] VASS, J. How Discord handles two and half million concurrent voice users using WebRTC. https://discord.com/blog/how-discord-handles-two-and-half-million-concurrent-voice-users-using-webrtc, September 2018. Accessed 31 July 2023.

[63] VON HIPPEL, M., MCMILLAN, K. L., NITA-ROTARU, C., AND ZUCK, L. D. A formal analysis of karn's algorithm. In *International Conference on Networked Systems* (2023), Springer, pp. 43–61.

[64] VON HIPPEL, M., VICK, C., TRIPAKIS, S., AND NITA-ROTARU, C. Automated attacker synthesis for distributed protocols. In *Computer Safety, Reliability, and Security: 39th International Conference, SAFECOMP 2020, Lisbon, Portugal, September 16–18, 2020, Proceedings 39* (2020), Springer, pp. 133–149.

[65] WANG, J., ZHANG, S., AND CHEN, F. Modeling and verification of sctp association management based on colored petri nets. In *2008 ISECS International Colloquium on Computing, Communication, Control, and Management* (2008), vol. 1, IEEE, pp. 379–383.

[66] WU, J., WU, R., XU, D., TIAN, D. J., AND BIANCHI, A. Formal model-driven discovery of bluetooth protocol design vulnerabilities. In *2022 IEEE Symposium on Security and Privacy (SP)* (2022), IEEE, pp. 2285–2303.

[67] YLONEN, T., AND LONVICK, C. The secure shell (SSH) transport layer protocol. https://www.rfc-editor.org/rfc/rfc4253, january 2006.

[68] ZOU, J., UYAR, M. Ü., FECKO, M. A., AND SAMTANI, S. Throughput models for SCTP with parallel subflows. *Computer Networks 50*, 13 (2006), 2160–2182.

8 Appendix

8.1 History of CVE-2021-3772

The vulnerability reported in CVE-2021-3772 arose from the following text in RFC 4960 [54]:

> If the value of the Initiate Tag in a received `INIT` chunk is found to be 0, the receiver MUST treat it as an error and close the association by transmitting an `ABORT`.

As illustrated in Figure 3, if an implementation did not check the validity of the `INIT` before transmitting an `ABORT`, then the RFC allowed for a DoS attack where the attacker would transmit an invalid `INIT` and thus trigger the victim to close an otherwise valid association. This vulnerability was first reported in the SCTP mailing list [26] and then reported in CVE-2021-3772 [48]. The vulnerability was subsequently patched in the Linux implementation by swapping the order of operations, to ensure the vtag is always checked before the itag [37]. The RFC was updated in 9260 [57] to say:

> If the value of the Initiate Tag in a received `INIT` chunk is found to be 0, the receiver MUST silently discard the packet.

and FreeBSD [4] implemented this patch when it was updated to reflect the new RFC.

8.2 User Model

We model not only each peer but also the user controlling each peer. The user commands are User_Assoc, User_Abort, and User_Shutdown. A user and its peer are synchronously connected. A subtlety of this composition is that the user is blocked from issuing unexpected commands, such as, a User_Assoc when the peer is already in Established. When the user issues User_Assoc or User_Shutdown the peer responds as shown in Figure 5. When the user issues User_Abort, the peer transmits an `ABORT` and transitions to Closed. The user is only allowed to issue a User_Abort when the peer is in an active association. The user herself is modeled nondeterministically with a single state, and a self-loop to and from that state to send each user command.

8.3 Erratum to RFC 9260

Beyond resolving the ambiguity described in Section 3.3, we have several minor suggestions for places the SCTP RFC could be made more clear through an erratum. First, we suggest incorporating the self-loop at Closed that occurs upon receiving a `SHUTDOWN_COMPLETE`, into the State Association Diagram in Section 4, closing the active/active teardown routine. Second, we suggest incorporating "initialization collision" (active/active establishment) into that same diagram, since it is a supported flow of the establishment routine and the default for WebRTC. Third, we suggest expanding 5.2 to explain how to handle other unexpected chunks, e.g., `COOKIE_ERROR`, `SHUTDOWN_COMPLETE`, etc., as such messages could be used by an Evil Server to deadlock a poorly implemented peer.

```
chan attacker_mem = [2] of {
    mtype:msgs,
    mtype:tag,
    mtype:tag,
    byte
};
active proctype attacker_replay() {
  mtype:msgs b_0;
  mtype:tag b_1, b_2;
  byte b_3;
  do
  :: atomic {
    AtoB ?? <b_0, b_1, b_2, b_3>
    -> attacker_mem ! b_0, b_1, b_2, b_3; }
  :: atomic {
    attacker_mem ?? b_0, b_1, b_2, b_3
    -> AtoB ! b_0, b_1, b_2, b_3; }
  :: atomic {
    attacker_mem ?? b_0, b_1, b_2, b_3; }
  :: break
  od
}
```

Figure 14: The replay attacker gadget automatically synthesized by KORG.

8.4 Performance

We time our experiments and patch verification tasks, and almost always, KORG terminates in seconds or minutes, with one interesting exception. In the Off-Path experiments, KORG takes about two hours to confirm that no attacks exist against ϕ_8, and about an hour and a half to find the CVE attack against ϕ_9. Recall that ϕ_8 and ϕ_9 are identical, except that the peer roles are reversed. Further inspection reveals these two properties are the largest in our property set, and the Off-Path attacker model is the largest attacker model, as it includes four processes (two peers, an attacker, and a channel) whereas the others involve fewer. The reason these two analyses take much longer than the others naturally follows, as KORG reduces to LTL model-checking, the runtime of which is polynomial in the size of the model and $O(\log^2|\phi|)$ in the size of ϕ [61]. This is further highlighted by the fact that the patch verification tasks terminate in a few minutes – and all the patch does is remove a transition from the code, reducing the model size. We report all run-times in Table 3.

	Off-Path		Evil-Server		Replay		On-Path	
	E	P	E	P	E	P	E	P
ϕ_1	2:20	2:13	0:23	0:23	0:3	0:3	0:15	0:15
ϕ_2	8:43	11:14	0:21	0:21	0:2	0:2	0:26	0:26
ϕ_3	3:20	12:53	0:20	0:20	0:2	0:2	0:25	0:25
ϕ_4	1:45	1:26	0:11	0:11	0:2	0:2	0:14	0:14
ϕ_5	2:57	1:35	0:10	0:10	0:2	0:2	0:12	0:12
ϕ_6	3:19	18:8	0:20	0:20	0:2	0:2	0:25	0:25
ϕ_7	1:43	4:41	0:11	0:10	0:2	0:2	0:13	0:14
ϕ_8	123:42	7:7	1:6	1:7	0:2	0:2	1:34	1:34
ϕ_9	86:10	6:48	1:5	1:5	0:2	0:2	0:11	0:11
ϕ_{10}	0:4	0:4	0:3	0:4	0:2	0:2	0:4	0:4

Table 3: Time taken (min:sec) to perform each (E) experiment and (P) patch verification on a 16GB M1 Macbook Air.

```
// Property 1
G((st[0] == Closed) -> (X(F(st[0] == Closed || st[0] == Established || st[0] == CookieWait))))
// Property 2
G(F(st[0] != ost[0] || st[1] != ost[1] || (st[0] == Closed && st[1] == Closed) ||
    (st[0] == Established && st[1] == Established)))
// Property 3
G((st[0] != ost[0] && ost[0] == ShutdownAckSent) -> (st[0] == Closed))
// Property 4
G(F(st[0] != CookieEchoed || timers[0] == T1_COOKIE))
// Property 5
G(st[0] != ShutdownReceived || st[1] != ShutdownReceived)
// Property 6
G((st[0] != ost[0] && ost[0] == ShutdownReceived) -> (st[0] == ShutdownAckSent || st[0] == Closed))
// Property 7
G(st[0] != CookieEchoed || st[1] != ShutdownReceived)
// Property 8
G((ost[1] == Established && ost[0] == Closed && everAborted == false && everTimedOut == false &&
    ost[0] != st[0]) -> (st[0] == Established || st[0] == IntermediaryCookieWait))
// Property 9
G((ost[0] == Established && ost[1] == Closed && everAborted == false && everTimedOut == false &&
    ost[1] != st[1]) -> (st[1] == Established || st[1] == IntermediaryCookieWait))
// Property 10
G((ost[0] == Established && (st[0] == ShutdownSent || st[0] == ShutdownReceived)) -> F(st[0] == Closed))
```

Figure 15: Our ten LTL properties are formulated in PROMELA as follows. We define our atomic propositions as follows in PROMELA, where `st` holds the state of each peer, `ost` holds the prior one, and `timers` holds the peers' timers.

Threat Model	P	Q	I/O
Off-Path	USERA ‖ PEERA ‖ CHANNEL ‖ PEERB ‖ USERB	Empty Program	Sends invalid messages
Evil-Server	USERA ‖ PEERA ‖ CHANNEL ‖ USERB	PEERB	Sends and receives valid messages
Replay	USERA ‖ PEERA ‖ CHANNEL ‖ PEERB ‖ USERB	CHANNEL	Sends messages previously received
On-Path	USERA ‖ PEERA ‖ PEERB ‖ USERB	CHANNEL	Sends and receives valid messages

Table 4: KORG inputs for each attacker model. P is the invariant component, which the attacker cannot change. Q is the variant component, which the attacker can prefix with a finite sequence of communication events. In the Off-Path attacker model, Q is the empty program, because the Off-Path attacker does not attach itself to any pre-existing system component. I/O specifies what the attacker is allowed to receive or send. ϕ will be one of the properties in §3.5, and the peer model (run by PEERA and PEERB) is described in Section 3.2.