UCLouvain

epl

École polytechnique de Louvain

Verifying protocol plugins

Author: **Nicolas Rybowski**
Supervisors: **Olivier Bonaventure, Axel Legay**
Readers: **Quentin De Coninck, Guillaume Maudoux**
Academic year 2020–2021
Master [120] in Computer Science and Engineering

Acknowledgments

This master's thesis was an interesting journey and, for this, I would like to thank the people who helped in its elaboration.

Of course my supervisors, Professor Olivier Bonaventure and Professor Axel Legay, who proposed this subject and helped me all along this project.

Florentin Rochet, Quentin De Coninck and Maxime Piraux for their patience regarding my questions on `PQUIC`*. Also, in general, all the INL members for the interesting ideas exchanges we had on the topics covered in this document.*

Quentin De Coninck and Tom Rousseaux for their precious reviews during the writing of this document.

Due to the various Open-Source projects used in the artifacts coming along with this thesis, I would like to thank the Open-Source community for its effort to provide freely (as in freedom) quality tools.

Last but not least, I would like to thank my family and friends for their support throughout this journey, especially during the tough times. A special thanks to Tom, who supported me as a researcher and as a friend, and Charline.

Contents

1 Introduction **1**

2 State of the art **3**
2.1 PQUIC . . . 3
2.1.1 Protocol overview . . . 4
2.1.2 Plugin verification and secure distribution . . . 6
2.2 BPF zoo . . . 8
2.3 Software verification . . . 9
2.4 Related work . . . 10
2.4.1 Static verification of eBPF programs. . . . 10
2.4.2 Securing software supply chains . . . 10
2.5 Conclusion . . . 10

3 Secure Plugin Management System **11**
3.1 Initial design choices . . . 11
3.2 Plugin Validator . . . 13
3.2.1 PV core micro-services . . . 14
3.2.2 Verifiers . . . 17
3.3 Plugin Repository . . . 17
3.4 Prototype evaluation . . . 18
3.4.1 Merkle tree build . . . 18
3.4.2 Trust proof validation . . . 19
3.4.3 Implementation size . . . 20
3.5 Deployment considerations . . . 20
3.6 Future work . . . 22
3.7 Conclusion . . . 24

4 Side effects property verification **26**
4.1 SeaHorn and annotated code . . . 27
4.1.1 SeaHorn and regular C code . . . 27
4.1.2 SeaHorn and PQUIC . . . 28

4.2 Methodology . . . 32
4.2.1 SeaHorn tuning . . . 32
4.2.2 Model testing . . . 33
4.3 PQUIC modeling . . . 35
4.3.1 Memory separation . . . 36
4.3.2 `PQUIC` structures . . . 37
4.3.3 Practical SeaHorn limitations . . . 38
4.3.4 Semantic constraints . . . 43
4.3.5 Working hypothesis . . . 44
4.4 Experimentation results . . . 46
4.5 Future work . . . 48
4.5.1 Current `PQUIC` verification improvements . . . 48
4.5.2 Pluginized protocols design improvements . . . 51
4.6 Conclusion . . . 53

5 Conclusion **54**

A Minimal examples of verification using SeaHorn **i**
A.1 Structure initialization . . . ii
A.2 Double pointer . . . vii
A.3 Legitimate `unsat` on `sassert(0)` . . . x
A.4 `UTHash` hashmap . . . x

B `PQUIC` structures **xiii**
B.1 `picoquic_packet_context_t` . . . xiv
B.2 `picoquic_sack_item_t` . . . xiv
B.3 `picoquic_packet_t` . . . xiv
B.4 `picoquic_cnx_t` . . . xv
B.5 `picoquic_tp_t` . . . xv
B.6 `picoquic_path_t` . . . xvi
B.7 `picoquic_connection_id_t` . . . xvi

C `PQUIC` memory helpers misuse **xvii**
C.1 `cnx_ro_replace` . . . xviii
C.2 `cnx_rw_replace` . . . xviii
C.3 `cnx_rw_pre` . . . xix
C.4 `plugin_rw_replace` . . . xix
C.5 `cnx_rw_tested_replace` . . . xx

Chapter 1

Introduction

Internet ossification The Internet we use daily relies on protocols like `OSPF` [43] and `BGP` [45] to create routes between routers and `TCP` to transport packets from endpoints to endpoints. However, those standardized protocols are old. The current `OSPFv2` specification has been defined in 1998 and the first drafts date from 1989. It is even worse for `TCP` whose first specification date back to 1974 [12].

These protocols inevitably needed to adapt to the always evolving network requirements due to the Internet expansion. It appeared that it is not an easy task, two main factors have to be taken into account.

First, networks operators rely on heterogeneous hardware environments. They differ by the devices types and vendors. This diversity is required to allow resilience in case of vendor specific bugs or simply to avoid vendor lock-in. It implies that standardized protocols are *required* to make all these devices talk with each others. This leads to the second issue.

Second, standardizing protocols is a slow process. According to Wirtgen et al. [49], the median delay required for an RFC publication is about 3.5 years. They show that it is even longer for some features which required up to ten years to be standardized. Even worse, those delays ignore the time gap between the initial idea and the submission of the first draft to the IETF.

This leads to an issue called the "protocol ossification". Computer networks loose in flexibility due to the difficulties to update network protocols.

Pluginized protocols To counter this issue, researchers proposed a major paradigm shift with `PQUIC` [25] and `xBGP` [49]. Rather than having monolithic and difficult to extend protocols, they consider that such protocols should be easily extensible *by design*. To do this, a pluginized protocol provides a small core exposing a flexible, vendor-neutral API. The latter allows network engineers to provide their own extensions called *plugins*. Now, if each network hardware vendor implements its own version of the protocol's core, plugins could be used to easily

propagate new features in the heterogeneous networks. Obviously, this extensibility raises one big issue.

Plugins security How could one trust protocol plugins injected in its devices? Is a given plugin malicious or legitimate? To answer those questions, the `PQUIC` authors [25] proposed a theoretical design for the *Secure Plugin Management System* (SPMS). This is a distributed system which has two tasks: (*i*) verifying that protocol plugins respect a given set of properties and (*ii*) distributing such plugins safely through networked peers. It produces trust proofs that are subsequently used by peers to validate that the plugins they receive are safe. Also, this offline verification method allows more uses-cases than verifying plugins on-the-fly right before a `QUIC` connection for instance. One could now consider using advanced tools to verify complex properties since time is not an issue.

This is the context of this master's thesis. It makes two main contributions to the protocol plugins verification field. First, it proposes a practical design and implementation of the SPMS. Second, it introduces the verification of a new property, called *side-effects*, on `PQUIC` plugins. This property ensures that a plugin only modifies connection fields for which it has authorization to do so. It extends the verification work begun by De Coninck et al. [25], who proved the termination of the majority of the `PQUIC` plugins.

The rest of this document is structured as follows:

Chapter 2 introduces the reader to the different state-of-the-art technologies tackled in this thesis. Among others, we cover `PQUIC`, the SPMS and some verification terminology.

Chapter 3 details our practical design and implementation of the SPMS. We also discuss some limitations of the current prototype and future works to fix them.

Chapter 4 outlines a new property we verify on `PQUIC` plugins. Notably, we prove formally that plugins only affect connection fields for which they have the right to do so. We detail in this chapter how we leveraged the SeaHorn framework [30] to verify this property. We also address some limitations encountered and propose future works to improve the current state of our work.

All the source code mentioned in this work is Open-Source and released on GitHub. The SPMS prototype is available at `https://github.com/nrybowski/SPMS` while the formal verification of the side-effects property on `PQUIC` plugins is available at `https://github.com/nrybowski/pquic-formal-model`.

Chapter 2

State of the art

Throughout this document, we will discuss issues and solutions about cutting-edge technologies which emerged from the most recent research around networking and program verification. This chapter introduces the reader to those technologies. In particular, we will first overview the new pluginized `PQUIC` protocol [25] which is at the core of this work. The Secure Plugin Management System (SPMS), a distributed system solving the verification and distribution issues for protocol plugins, is also introduced. We then present `eBPF`, a platform-independent assembly language leveraged by `PQUIC`. Next, basic software verification concepts are outlined to the reader. Finally, we cover some related works on the aforementioned technologies.

2.1 PQUIC

`QUIC` is a recent transport protocol which encrypts almost the whole packet's content, contrariwise to the well-established `TCP`. Pluginized `QUIC` (`PQUIC`) [25] is a new pluginized protocol based on `QUIC`. It allows `PQUIC` clients and servers to exchange *protocol plugins*. Those are platform-independent, user-provided bytecodes allowing dynamically rewriting behaviors of the protocol. Hence, `PQUIC` allows customizing its function on a per-connection basis. Easily changing the behavior of a network protocol on a remote host solves the deployment issues currently encountered on the Internet. However, it obviously raises security issues, i.e. is a given plugin malicious or legitimate?

In this section, we will first cover how `PQUIC` runs such user-provided code. Then, we will describe a distributed system, proposed by the `PQUIC`'s authors, which solves the safety concerns.

2.1.1 Protocol overview

PQUIC is composed of three main elements: (*i*) a lightweight core implementing *protocol operations*, (*ii*) a *Plugin Runtime Environment* (PRE) which is a specific environment designed to execute the plugins and (*iii*) an API allowing plugins to interact with the protocol's core from within their PRE. The next points detail each of those components. We also provide more insights on the structure of protocol plugins before introducing the PRE.

Protocol operations The QUIC protocol has been summarized into elementary operations named *protocol operations*, equivalently denoted as *protoops* in the rest of this document. These are common routines being part of any QUIC implementation. Such operations are, for example, the update of the RTT or the ACK delay, the parsing of QUIC frames and so on. Together, they form the protocol's core.

They are represented as high-level specifications, similar to function signatures, defining the inputs, outputs and effects of an operation. Such specification is shown in Listing 2.1. PQUIC provides, for most of them, a default implementation defined by the 14th version of the IETF QUIC drafts [35].

```
/**
* Update the ack delay used locally based on the latest rtt estimate
* \param[in] pkt_ctx \b picoquic_packet_context_t* The packet context
* to update
* \param[in] old_path \b picoquic_path_t* The path on which the RTT
*  estimate was computed
* \param[in] rtt_estimate \b int64_t The RTT estimate computed on the
*  given path
* \param[in] first_estimate \b bool Indicates whether the RTT
* estimate was the first one for the given path
*/
#define PROTOOPID_NOPARAM_UPDATE_ACK_DELAY "update_ack_delay"
extern protoop_id_t PROTOOP_NOPARAM_UPDATE_ACK_DELAY;
```

Listing 2.1: `update_ack_delay` protoop definition

Protoops also define insertion points, called *anchors*, in the protocol's core. There are three of them. *Pre* and *post* anchors allow attaching user-defined code respectively before and after a given protoop. A *replace* anchor allows replacing the default core's implementation of a given protoop.

Protocol plugins The user-defined code called "plugin" is composed of two elements: one or multiple *pluglet(s)* and a *manifest*. A *pluglet* is a bytecode implementing a protoop at a specific anchor. The *manifest* is a file containing metadata such as the plugin name and the anchors at which a pluglet has to be attached. This is illustrated by Listing 2.2.

```
be.mpiraux.ack_delay
update_ack_delay replace update_ack_delay.o
is_ack_needed replace is_ack_needed.o
```

Listing 2.2: `ack_delay` plugin's manifest

In this simple example, there are two pluglets: `update_ack_delay.o` and `is_ack_needed.o`. They are both attached to the "replace" hook of their respective protoop. The only additional metadata mentioned is the plugin's name.

There also exists a specific *include* keyword allowing reusing a previously defined plugin to define a new one, as shown by Listing 2.3.

```
be.qdeconinck.multipath.rtt negotiate
multipath_cond.plugin include
schedule_path replace path_schedulers/schedule_path_rtt.o
```

Listing 2.3: *include* illustration

This way, plugins may be combined and incrementally built. That being said, this requires some careful monitoring during the injection step in order to avoid circular dependencies. The same concern raises while considering pluglets calling themselves other pluglets. Because of the combinatorial state explosion issue, `PQUIC` handles this problem at runtime by keeping track of all running protoops. If a call to an already running protoop is detected, the connection is stopped with an error.

Plugin Runtime Environment (PRE) In order to ensure that plugins are platform independent, and thus easily deployable at large scale, a specific execution environment is required. It is also a key aspect in the security issue raised by the execution of user-defined code.

Among different solutions, De Coninck et al. [25] chose to work on a modified version of the `uBPF` VM [5]. Originally, as it runs in user-space, it does not require a verifier as its Linux kernel `eBPF` parent implementation. However, the `PQUIC`'s authors added a relaxed version of the `eBPF` verifier, allowing more various use-cases. It also contains some monitoring capabilities similar to Software-based Fault Isolation. Among others, specific instructions are added in the JITed `eBPF` plugin's bytecode to monitor the memory accesses. If an out-of-bound access is noticed, the plugin is removed and the connection is terminated.

Besides those safety insurances, a PRE provides a strong memory isolation between the protocol's core and the plugins. A PRE is created for each injected pluglet. It contains a dedicated stack and registers. It also provides a heap common to all the pluglets of a plugin, allowing pluglets to share data between them. As the plugins are injected on a per-connection basis, the PRE exposes a connection context to each pluglets.

API The real glue between the protocol's core and the PREs is the API. `PQUIC` exposes many helper functions to plugins in order to let them interact with the core. They are six main families of helpers.

First, **getter/setters** allow pluglets to interact with the connection context. It also abstracts the inner `PQUIC` implementation from plugins. Hence, a specific plugin could work on multiple `PQUIC` versions.

Second, **plugin memory** helpers provide control to pluglets on the plugin's heap, which is stored in the host memory. They abstract memory allocation and free routines.

Third, the API exposes functions for **sharing data between pluglets**. They implement a communication channel by assigning identifiers to specific plugin's memory areas, stored in the host memory. When a pluglet allocates such data, other pluglet may then access it by using this identifier.

Fourth, access to **memory areas external to the PRE** is enabled by wrappers around `memcpy` and `memset` functions. They are required, for instance, when a plugin has to write new `QUIC` frames.

Fifth, a pluglet may **call another protoop** in its implementation. Such helpers are provided to monitor pluglets calls for circular dependencies as previously mentioned.

Sixth and last helper family, the API provides a way for pluglets to **schedule the transmission of `QUIC` frames**.

2.1.2 Plugin verification and secure distribution

Plugins are provided by developers and they have full control on the pluginized protocol behavior. Hence, executing unverified code injected by a remote host raises important security issues [1]. Malicious pluglets could have dramatic impacts on the privacy of end-users data. However, verifying plugins on-the-fly just before their injection strongly reduces the amount and diversity of validation technique one could leverage. Indeed, verification tools taking more than a few tens of milliseconds would slow down the `PQUIC` connections beyond an acceptable point.

To tackle this problem, De Coninck et al. [25] proposed a distributed system called *Secure Plugin Management System* (SPMS). It allows offline plugin verification and trust proof distribution among networked peers. This SPMS contains two main components: a *Plugin Repository* (PR) and multiples *Plugin Validators* (PV). They are both detailed in the next points. One may also consider plugins developers and protocol peers as part of this system.

[1] Interestingly, it is currently what we all do when visiting Internet websites including JavaScript scripts.

PV This component embeds one or multiple verifiers, each asserting that plugins respect a specific property, e.g. termination. It builds, at regular intervals, a Merkle tree [42] including the bindings of all successfully verified plugins. The root of the tree is then signed to form a Signed Tree Root (STR).

The plugin validation process on the pluginized host side is multi-step. It begins by querying a PV for its STR, the plugin's binding and the path to reach the plugin's binding in the PV's Merkle tree. Then, the host recomputes the plugin's binding hash. With the binding's hash and the tree path, it recomputes the tree's root hash. Finally, the root hash is compared with the one stored in the STR. If they are equals, then the plugin has been successfully verified by the PV. If they are different, the plugin's bytecode received is not verified by the PV.

PR This component is the central point of the system as it puts each system's component in contact. Its main roles are (*i*) collecting the source code of each plugin pushed by plugin developers, (*ii*) distributing the plugins to the PVs for verification and collecting the verification result for each of them and (*iii*) storing the trust proof (the PV's STR) from each PV in the system.

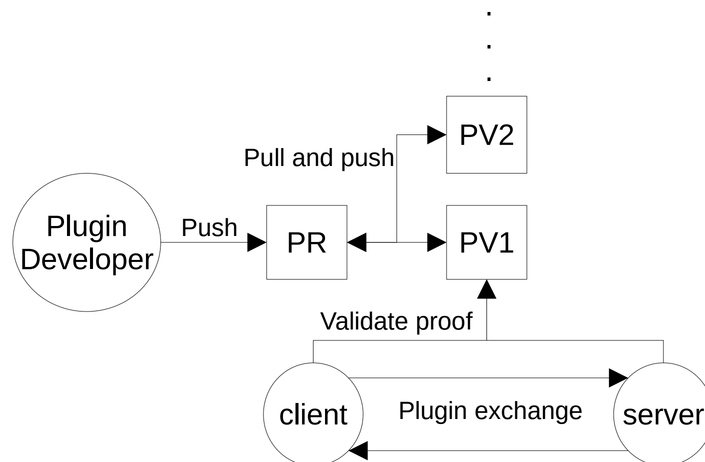


Figure 2.1: Secure Plugin Management System structure.

Figure 2.1 illustrates the SPMS structure and its workflow. First, plugin developers push their plugins to the PR. The PR then notifies each PV that new plugins are available for verification. PVs pull the new plugins and perform their verification. Eventually, PVs will produce their Merkle tree and push their STRs to the PR. If a verifier failed, the verification logs are also pushed to the PR.

On their side, the pluginized peers exchange plugin bytecodes. Upon plugin reception, they execute their validation process. If it is successful, the plugins are injected and the connection starts.

Plugin developer also have to interact with the PVs to ensure that the validated plugins are actually the ones they sent.

2.2 BPF zoo

cBPF In the late nineties, multiple Unix versions provided facilities for packet capture and later user-space processing. The main functionality provided by such tools is the packet duplication from the kernel-space to the user-space. In order to reduce the overhead caused by the duplication of packets later filtered out in the user-space, an efficient kernel-space packet filter, namely `BSD Packet Filter` (BPF) [40], has been created. This technology, now known as `classical BPF`, allowed tools like `tcpdump` to present the performances we currently know by filtering out undesired packets in the kernel [40].

eBPF While `cBPF` allows injecting simple filters in the kernel for packet selection, having a programmable kernel interested the kernel community. Many uses-cases requiring user-provided code to dynamically modify the kernel behavior arise.

This is the origin of `extended BPF` (`eBPF`), which is an in-kernel Virtual Machine using its own instruction set[2]. The kernel exposes various hooks[3] to which `eBPF` programs may be attached. An `eBPF` program is able to provide information to user-space programs through multiple channels[4].

In the notable projects leveraging `eBPF` outside the networking context, we may cite *seccomp* [10]. It uses `eBPF` programs to allow administrators to set fine-grained access rules for kernel syscalls.

A critical aspect of `eBPF` programs is safety since they can modify on-the-fly the kernel's behavior. To this end, multiple hardening and memory abstraction[5] techniques are used to ensure isolation between the `eBPF` VM and the kernel-space [3]. They also provide protections against kernel corruption attempts. However, the very first defense line is the in-kernel verifier.

eBPF verifier To ensure that a program is safe to run on the VM, the verifier applies several verification on its bytecode. Among others, the verifier asserts that a given program will end[6], has a return instruction and does not try to access restricted memory areas. Accesses to kernel memory areas are limited by provided

[2] `eBPF` bytecode might be JIT'ed to reach near native performance.

[3] Among others, syscalls, any kernel function, tracepoints, specific hooks in the network stack known as *sockops*, etc.

[4] See `https://blogs.oracle.com/linux/notes-on-bpf-3`

[5] In reality, some memory accesses are re-written by the verifier during its verification.

[6] A limited support of loops has been recently added but is still limited [47].

helpers. Also, the accepted program size, in term of `eBPF` instructions, has a hard limit which means that such program may not be arbitrarily long nor complex. Indeed, the verifier explores each possible execution path. If it is not able to end this task within its configured bounds, it denies the program injection. Even if the liberty left to the developer seems to be quite narrow, the `eBPF` community is working to improve this technology.

uBPF While the main implementation of the `eBPF` VM lies in the Linux kernel, the `uBPF` project [11] implements the VM for user-space usage. This version comes with a simpler embedded verifier which opens a lot of new possible usages for this general usage VM.

2.3 Software verification

Software verification is the act of *ensuring that a program conforms to its specifications*. The usual way of doing so is by testing the program. That is, the developer provides input data to the program and observes the outcome. By using specific inputs triggering code's corner cases, this method may prove the existence of errors but not their absence. Indeed, only a subset of the possible program executions are explored, which is a common issue of such dynamic verification.

On the other hand, there exists static verification methods relying on software inspection. They can prove the absence of errors by exploring all the program executions but may also produce incorrect results.

In fact, software verification is a balance between soundness and completeness [23]. A verification method is said *sound* when it correctly states that a program has no errors. In other words, it does not produce *false negative* outcomes. On the contrary, a verification method is said *complete* if each error it finds is a real error. That is, it does not produce *false positive* results. Soundness is possible with static analysis but completeness requires real program execution.

There exist many static verification methods but the one exploited by the tools leveraged in this document is called *Bounded Model Checking* (BMC) [20]. A complete overview of this method is beyond the scope of this document. In a nutshell, an abstract representation of the verified software (the model) and a property are provided to the Model Checker. The negation of the property is checked up to a fixed depth k in the state space of the possible executions. This is reduced to a SAT problem, which is the satisfiability of a propositional Boolean expression. If the negation of the property is satisfied then an error is found and a counterexample trace is produced. Else, the property holds for all the explored states.

2.4 Related work

In this section, we review some latest works related to the aforementioned topics. We will especially cover projects covering the verification of `eBPF` programs and the securing of software supply chains.

2.4.1 Static verification of `eBPF` programs.

The current limitations of the Linux kernel `eBPF` verifier are well-known [4]. Among others, the size of accepted programs is strongly limited and loops are not yet fully supported [47]. Given the growing popularity of the `eBPF` kernel implementation[7], they strengthen the need for an efficient and accurate verifier.

To this end, Gershuni et al. introduced `PREVAIL` [29], a static verification tool for `eBPF` programs. It is designed to be sound but not complete. Their framework leverages Abstract Interpretation on a new language (`eBPFPL`) they introduce to abstract `eBPF` programs.

2.4.2 Securing software supply chains

A software supply chain is the whole path between the software developers and the end-users [36]. It includes, among others, the software source code, its dependencies, the build process and binaries distribution. Each of those chain links are subjects to attacks, hence securing the whole supply chain is crucial.

While the SPMS [25] is designed to secure protocol plugins distribution, other projects emerge to protect whole open-source software [15, 13]. One of them is *sigstore* [14], a Linux foundation project. It leverages existing technologies such as X509 PKI [21] and Certificate Transparency [38]. The SPMS and sigstore are quite similar as they both use Merkle trees and certificate signature to distribute trust proofs. However, the two approaches have a major difference, sigstore relies on an external monitoring entity while the SPMS is self-sufficient. Also, sigstore focuses on signing release artifacts such as tarballs, binaries and container images.

2.5 Conclusion

In this chapter, we covered the `PQUIC` pluginized protocol and the SPMS. This distributed system is designed to solve the verification and distribution issues of user-defined plugins. We then introduced the `{c,e,u}BPF` projects leveraged in current pluginized protocols. We also outlined some basic concepts of software verification. The chapter is finally closed by related works.

[7] See `https://ebpf.io/projects`

Chapter 3

Secure Plugin Management System

This chapter describes our prototype implementation of the *Secure Plugin Management System* (SPMS) defined by De Coninck et al. [25]. A detailed overview of this system, as thought by the `PQUIC`'s authors, is presented in Chapter 2. The source code of our prototype is released as an open-source project on GitHub[1].

The chapter first presents a high-level practical design overview of the SPMS. It then provides detailed descriptions of the Plugin Validator (PV) and Plugin Repository (PR) internal architectures. They are followed by the evaluation of specific points of the prototype. Next, we discuss considerations about the system deployment. Finally, we close the chapter with some future work.

3.1 Initial design choices

Two main elements guided our practical design and implementation of the SPMS prototype: (*i*) the architecture model and (*ii*) the communication requirements between the SPMS elements. This section introduces both points.

Architecture model The main constraint for the SPMS practical design is its distributed nature. It requires modularity due to the number of entities in the system. Indeed, the number and type of verifiers running in a PV are not strictly defined. Furthermore, they are expected to evolve with the software verification research field. Hence, the internal designs of PVs and the PR have to be flexible enough to easily integrate new verifiers and additional services over time. Some external tools, like compilers, should also easily integrate the system. These

[1] `https://github.com/nrybowski/SPMS`

considerations, with the distributed nature of the SPMS, led to the choice of a microservice-based architecture.

Practically, the main functionalities of a SPMS component are isolated in their own Docker container [41], each presenting a REST API allowing inter-services communication. This choice avoids monolithic implementation and thus allows easily integrating new services according to requirements updates. Furthermore, it also facilitates the integration of scaling policies that could be required for systems under heavy load.

Inter-components communication The communication between the PR and the PVs is another fundamental point of the SPMS. The PR must be able to send notifications about plugins available for verification to many PVs, each embedding one or multiple verifiers for a specific pluginized protocol. Furthermore, the administrator of the system is not expected to know the number and type of PVs at deployment time. PVs may be added or updated during the system's lifetime and the PR has to be able to manage such changes.

The *Publish-Subscribe* messaging pattern is particularly well suited for such requirements as it decouples the senders from the receivers by introducing a topic-based messaging system. Publishers (senders) send their messages for specific topics without knowing which subscribers (receivers) are listening to them. A common way to implement such system is by using a middleware, called *broker*[2], to which the publishers send messages and subscribers register to listen for specific topics. `MQTT`, a lightweight publish-subscribe protocol running over `TCP` [16], is leveraged for this SPMS implementation. In particular, we use the *Mosquitto* broker [39] and the *Paho* `MQTT` python library [27], both being open-source projects from the Eclipse Foundation. For simplicity, the `MQTT` broker is embedded in the PR architecture but it may also be considered as an additional component of the SPMS if required.

By default, `MQTT` does not provide reliable security insurance other than simple plain-text authentication and authorization capabilities. That being said, authentication of a PV to the PR is required to ensure that no spurious STRs are introduced in the trust system. In order to allow mutual authentication with mutual `TLS` (`mTLS`) [46], a Public Key Infrastructure (PKI) [21] is deployed. That is, a root Certificate Authority (CA) is created for the whole SPMS and is used to sign `SSL/TLS` X.509 certificates [21] for each component. Those certificates are then used for secure `MQTTs` and classical `SSL/TLS` connections between the PVs and the PR.

[2]`https://mqtt.org/software/`

3.2 Plugin Validator

A PV contains two main components: (*i*) the micro-services forming its core and (*ii*) the embedded verifiers. Figure 3.1 shows a simplified overview of the internal PV composition. On one hand, the core is common to every PVs regardless of the verification performed, and is protocol agnostic. On the other hand, the verifiers are property and/or protocol specifics. They are not micro-services by themselves since they do not expose REST APIs. They rather implement a simple routine which queries, at regular interval, the core for unverified plugins. Each new plugin is verified and the result of the verification is sent back to the core.

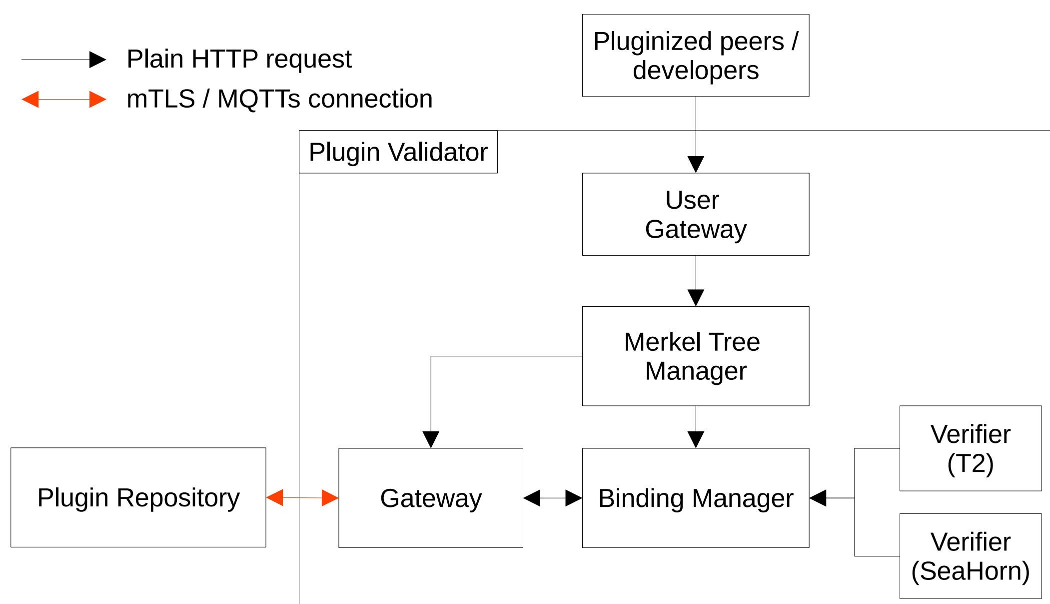


Figure 3.1: Plugin Validator simplified architecture with two embedded verifiers.

By default, a plugin is considered as valid, and consequently stored in the PV's Merkle tree, if it verifies all the properties of the PV. However, each PV embeds a unique expression in clausal normal form[3], e.g. $P_0 \wedge (\neg P_1 \vee P_2)$, whose literals are the properties to verify. This lets PVs administrators express a "higher-level" property verified by the whole PV.

Originally, this flexibility was thought to be implemented on the `PQUIC` peers side [25]. That is, a peer could express precise safety requirements with this kind of logical expression. They would query different PVs for their STR and recompute them. Then, they would evaluate the PVs combination. This approach could lead

[3] A conjunction of disjunction.

to an overhead for complex safety requirements, due to the multiple requests and STR re-computation.

We decided to implement this additional step into the PVs to reduce the amount of efforts that a peer should provide. This approach is not limiting since one could always deploy PVs embedding a single verifier. Once a pattern of regularly asked safety requirements is identified, the SPMS administrator could deploy a new PV gathering them in one place.

One could consider the hypothesis of a malicious PV. One advantage of this distributed system is that a pluginized host does not have to trust a single PV. The trust comes from the aggregation of multiple PVs outputs. However, we emphasize that our approach of gathering multiple properties in a single PV is not limiting. Let's imagine three PVs gathering the same five properties. It remains interesting for a pluginized host to only query three PVs rather than fifteen PVs with a single property. With our approach, both solutions are possible.

The next points of this section detail the services of Figure 3.1.

3.2.1 PV core micro-services

We first cover the two main services of the PV's core. They are charged to handle plugins data and the PV's Merkle tree. Together, they implement the PV's behavior as defined by the `PQUIC` authors [25]. In order to ensure their correct isolation from the outside world, we introduce two gateways. Each of them has specific security requirements, which explains why there is more than one. Finally, we discuss some additional services not illustrated in Figure 3.1.

Binding Manager This service stores all information known on every plugin processed by the PV. To this end, it maintains a volatile, in-memory, database collecting for each plugin: the compressed source code, its associated binding and the status for each verifier of the PV.

```
{
    "<plugin name>": {
        "binding": <compressed plugin source code>,
        "tar_binding": <plugin binding>,
        "verifiers": {
            "P0": {"status": "failure", "log": <failure log>},
            "P1": None,
            "P2": {"status": "sucess"}
        }
    }
}
```

Listing 3.1: Illustrative entry of the Binding Manager's internal database with three verifiers.

This database, illustrated by Listing 3.1, is implemented as a hashmap protected by a very simplistic read-write locking scheme[4] to allow safe concurrent accesses to the data.

Upon verifier request, the Binding Manager will send the first plugin for which the related entry is `None`. Once the verifier sends back its result, the latter is stored then parsed. If it is a failure, the verifier log is forwarded to the PR as specified by De Coninck et al. [25].

This service also implements a function which selects the plugins respecting the PV expression. `None` or failed verifiers status are evaluated to *False* while a success status is considered as *True*. Let's reconsider Listing 3.1, for example, the expression $(\neg P_0) \wedge (P_1 \vee P_2)$ will evaluate to *True* while $P_2 \wedge (P_0 \vee P_1)$ will evaluate to *False*. This means that the plugin should be included in the PV's Merkle Tree if it matches the first expression but not with the second.

Merkle Tree Manager This service is in charge of producing the PV's Merkle tree at regular intervals and providing access to the data stored within. A pseudo-code of its main behavior is provided in Listing 3.2. First, it requests to the Binding Manager the plugins respecting the PV's expression. They are then used to produce a fresh Merkle tree whose root hash is signed using the PV's private key [21]. This produces a Signed Tree Root (STR). Finally, this STR is stored and a copy is forwarded to the PR for archiving via an `HTTPS` request.

```
while(True):
    sleep(TIMEOUT)
    plugins = request_validated_plugins()
    merkel_tree.build(plugins)
    str = merkle_tree.generate_str()
    send_str_to_pr(str)
```

Listing 3.2: Pseudo-code for the Merkel Tree Manager service.

We provide a minimal and somewhat basic implementation of the Merkle tree structure as a python module, as well as some basic test cases. It is based on a hashmap representing the tree and the only hashing algorithm currently supported is `sha256`.

De Coninck et al. [25] introduce a special value called *epoch* which regulates the STR production pace and synchronize the whole system. For the sake of simplicity, the timeout defining the tree construction interval is arbitrarily defined to 10 seconds. Further thinking is still required to formally define the epoch value.

[4]Based on the `readerwriterlock` Python library. See `https://github.com/elarivie/pyReaderWriterLock`.

(PR) Gateway This service isolates the PV core from the outside world by being the only interface between the PV and the PR. No inner service is able to directly contact the PR and inversely.

Especially, this service ensures mutual authentication between the PV and the PR. On PV startup, it initiates a `MQTTs` connection toward the PR's broker and registers itself to its topics of interest, e.g. the protocol or the properties it verifies. Upon reception of a plugin availability notification, it fetches the plugin from the PR through a `mTLS` secured connection. It then forwards it to the Binding Manager. In the other direction, it forwards the verifiers logs (from the Binding Manager) to the PR in case of verification failure and the PV STRs (from the Merkle Tree Manager).

User Gateway De Coninck et al. [25] specify that plugin developers must be able to interact with PVs, among others, to verify the verification status of their code. An additional gateway is added between external users and the Merkle Tree Manager to ensure isolation of the PV internals from the outer world. Those users are indiscriminately plugin developers and `PQUIC` peers. Currently, it allows users to request a plugin binding, the path to reach this binding in the Merkle tree and the PV's STR. This is sufficient for a simple client to verify if a given plugin has been validated by the PV.

The difference between this service and the previous gateway resides in the kind of accesses. Indeed, the requests via the User Gateway may not modify the PV state, they are only lookup. Inversely, the (PR) Gateway is the only point allowing external modifications of internal PV and PR states, e.g. upon plugin reception on PV side or STR storage on PR side. Therefore, strong security and authentication insurances are required, which is not the case for the User Gateway.

Additional services To ease the comprehension of the PV basics, we ignored some additional services in Figure 3.1 but for the sake of completeness we rapidly cover them in this point.

Binding Generator In a first iteration of the prototype, only bindings (plugins bytecode) were sent to PVs. This was quite limiting for the diversity of tools usable to verify plugin's properties. In order to widen our verification abilities, we added this service to produce a plugin binding based on its source code. That is, the Binding Generator receives a plugin under the form of a `tar` archive containing the plugin source code, compressed with the `bz2` algorithm. It is generally composed of the pluglets source code, the plugin manifest and some additional header files. Once the content of the archive is extracted, all the files are compiled to produce `eBPF` bytecode by leveraging the LLVM compiler. Finally, the

binding is produced under the form of a new `tar` archive containing the pluglets bytecode and the plugin manifest.

This whole procedure is triggered on demand, upon plugin source code reception, by the Binding Manager. It then stores both plugin representations.

Logger A simple logging service which collects events from each service for debugging and tracing purpose.

3.2.2 Verifiers

Currently, two main properties verifiers are available for PVs deployment: (*i*) pluglets termination and (*ii*) pluglets side-effects limitation.

The first one is a pipe composed by the LLVM compiler, a LLVM bitcode to T2 format converter and the T2 temporal prover [22]. This verifier reproduces the results obtained by De Coninck et al. [25] so not many details are provided here. The majority of the pluglets have been proved to terminate. For some of them, the `PQUIC` authors had to slightly adapt the pluglet's source code to achieve this proof. Also, a minority of the pluglets could not be proven due to their complexity.

The second verifier allows asserting that a pluglet only modifies fields of its connection context for which it has the right to do so. This property, called *side-effects*, is deeply detailed in Chapter 4.

3.3 Plugin Repository

A minimalist Plugin Repository design, illustrated in Figure 3.2, is proposed and implemented. It is sufficient to enable communication with PVs but do not provide the full functionalities expected from a PR. This is due to the main focus on the PV development.

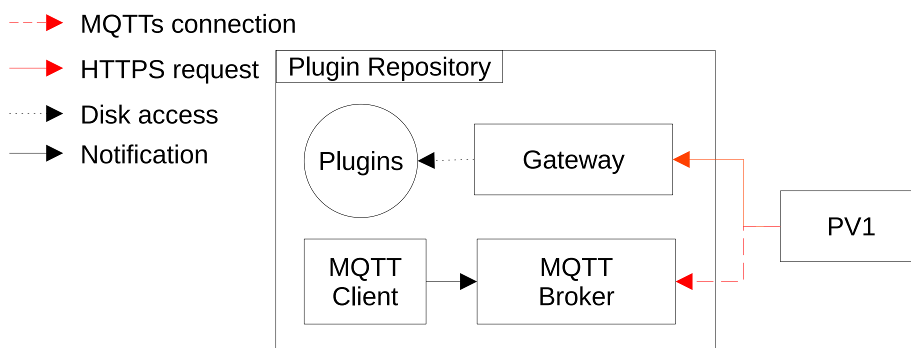


Figure 3.2: Simplified PR architecture.

This PR architecture consists of two services, an `MQTT` broker and a simple REST API gateway. The CLI `mosquitto MQTT` client is used to manually send notifications toward PVs through the broker. Upon reception of such notification, the PVs request the advertised plugins through the PR gateway. It returns the plugins under the form of a `bz2` compressed `tar` archive. Those archives are currently manually stored in the PR. Once a PV verified a plugin, it sends back to the PR the verification logs in case of failures and its new STR. Those are just displayed on the PR side for debugging purpose but are not yet stored anywhere.

3.4 Prototype evaluation

One of the most crucial aspect of the SPMS is the Merkle tree embedded in PVs. It is the structure allowing distributing trust proofs among pluginized peers. To validate its behavior, we did some performance measurements: (*i*) the time required to build the tree in a live SPMS deployment and (*ii*) the time that a pluginized host needs to ensure a plugin has been verified by a PV. The next points present the observed results. Then, the section is closed by an overview of the prototype size in term of Single Line Of Code (SLOC).

3.4.1 Merkle tree build

The Merkle tree is a perfectly balanced binary tree. This implies that the number of leaves equals $2^{depth-1}$ and the total number of nodes is $2^{depth} - 1$. Incrementing the depth of the tree by one doubles the number of leaves and thus the number of plugins the tree can contain[5]. However, it also doubles the whole tree size. Hence, different tree depths are compared to evaluate the overhead that this parameter introduces since it directly impacts the number of plugins the tree can embed.

Figure 3.3 illustrates the median duration required to build a PV's Merkle tree within a live SPMS deployment. The evaluation is performed as following. Dummy plugins are created by duplicating the `be.michelfra.disable_cc` one. Only the name is modified by appending an identifier composed of two digits. Consequently, the hashing duration of the binding is expected to be equivalent for all the dummies plugins. The plugins are then added in bursts within the system through the PR. It is deployed with a dummy verifier which returns a success status after 5 seconds. Once the system treated the plugins burst, 20 measurements are performed for each configuration.

[5] In reality, a leaf can contain more than one binding in case of hash collision. The counterpart is that it increases the hash computation cost and the quantity of collision hashes that a pluginized host has to fetch for the plugin trust proof validation. This is also true for the Merkle tree build process. Hence, having exactly one binding per leaf is the best case scenario.

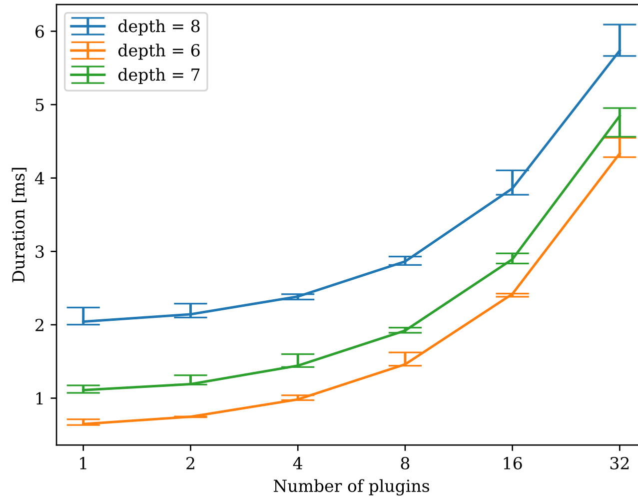


Figure 3.3: Merkle tree build time.

The three curves show that building a Merkle tree takes a few milliseconds, depending on its depth and the number of embedded plugins. The gap between the curves remain globally constant, hence they present a quite similar behavior. This indicates that incrementing the depth effectively increases the computation time, but no dramatic raise is observed. By predicting the number of plugins a PV should handle, one could choose an appropriate Merkle tree depth to ensure optimal build time.

3.4.2 Trust proof validation

The whole point of the SPMS is to offload the verification of plugins safety properties from pluginized peers. However, they still have to verify that a given plugin has been approved by the system. To this end, we implemented the validation process that a pluginized host should perform. It consists in three steps. First, the host requests to the PV the binding of a plugin, the path to reach this binding in the PV's Merkle tree and the PV's STR. Second, it recomputes the tree's root hash by using the binding hash and the tree path. Third, it extracts the root hash stored in the STR[6] and compares it with the one it previously computed.

Figure 3.4 shows the mean duration of the trust proof validation process. The experiments are run 20 times. The root hash computation takes, on average, 0.15 ms to complete while the STR decoding and hash comparison takes 0.68 ms. No significant variance is observed nor impact coming from the tree depth. However,

[6]By using the public key of the PV which is embedded in its publicly available SSL certificate.

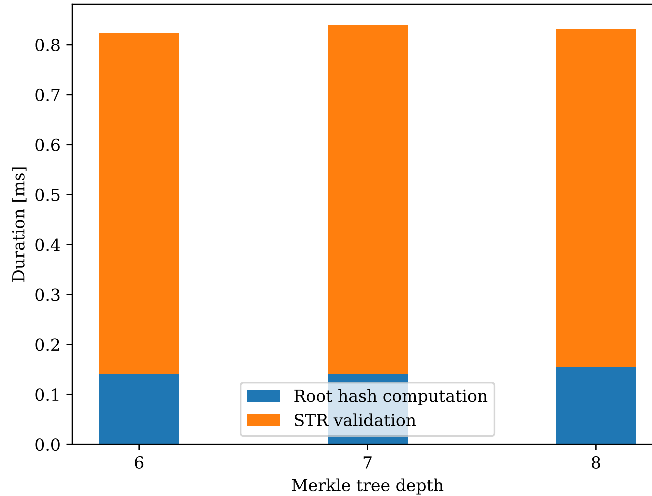


Figure 3.4: Trust proof validation timing.

the last step of the process takes almost 4.5 times more time than the root hash computation. Caching the root hash stored in the STR could dramatically reduce the time required by the hash comparison. Also, the timing of the first step is ignored in our measurements since it depends on the state of the network between the pluginized peer and the PV. That being said, it will be the real limiting factor of the validation process. Hence, caching the STR could also help to limit the number of requests to a PV that a peer should perform to validate a plugin.

3.4.3 Implementation size

The SPMS prototype is implemented with 845 SLOCs in Python language [44]. This number excludes the code required for its automated deployment. Table 3.1 details the amount of code per SPMS component.

3.5 Deployment considerations

In this section, we discuss some important deployment points. The technologies used for the prototype SPMS build process and deployment are first presented. We then detail an additional SPMS component we designed and implemented to ease the distribution of micro-services and verifiers inside the distributed system.

	Module	SLOC
PV	binding_generator	48
	binding_manager	314
	logger	18
	mt_manager	107
	pr_gateway	88
	user_gateway	17
PR	pr gateway	22
Verifiers	verifier controller	45
	pquic wrapper	29
	t2 wrapper	56
Utils		101
Total		**845**

Table 3.1: SPMS SLOC count per component.

Automated deployment technologies Each component of the SPMS is practically represented by a Linux Kernel-based Virtual Machine (KVM) [37] running its associated micro-services stack. Working with a multi-VM environment implies that its deployment must be easily reproducible. This motivated the usage of the Infrastructure as Code (IaC) *Terraform* [33] and *Packer* [32] open-source tools by Hashicorp. *Terraform* allows describing the expected state of the whole system in a configuration file. It is then used to build and deploy the required KVM VMs with the QEMU emulator [19] by using its *Libvirt* plugin [26, 34]. *Packer* is used to automate the build of the machine images required during the deployment. In order to keep the SPMS space efficient, the machine images have been designed as a layered stack.

Note that the current infrastructure configuration file is written for local deployment on a single KVM enabled host. However, thanks to *Terraform*, only a few changes are required to allow deployment of the SPMS on common Infrastructure as a Service (IaaS) cloud providers such as Microsoft Azure, Amazon AWS or Google Cloud[7]. That being said, the current prototype **should not** be deployed in production since its development has been focused on functionalities, not on security. Hence, it lacks some very basic hardening steps such as key-only `SSH` authentication and so on.

Build Pipe Each SPMS component embeds common services and one could consider rebuilding those containers on VM deployment. This approach implies to

[7] `https://registry.terraform.io/browse/providers?category=public-cloud&tier=official`

embed the source code of the services in the PVs and PR. This is not desirable notably because of size issue and build time duplication. An additional component, illustrated by Figure 3.5, is added to the SPMS to allow easy distribution of common services.

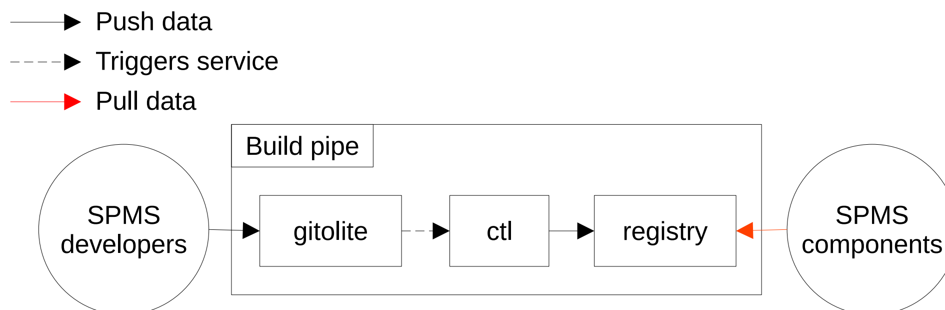


Figure 3.5: Build pipe architecture.

Its function is the following. First, SPMS developers push the services source code to the gitolite server [48] which is a simple git hosting server operating over `SSH`. Then, through git hooks, it triggers the controller (ctl) service which builds all the required containers. Finally, those are pushed to a container registry[8]. This allows to newly launched VMs to simply pull their services from the registry rather than rebuilding all the containers.

3.6 Future work

The current SPMS implementation is still a prototype. We propose in this section some future works based on known limitations of the system.

PR implementation The current PR implementation is meant to provide the minimal interactions with PVs. It does not respect the specifications planned by De Coninck et al. [25]. For instance, a plugin developer cannot easily add plugins source codes to the PR and it does not have persistent storage for the data sent back by the PVs. However, we propose in Figure 3.6 a full architecture for this component, which extends the one implemented.

A Version Control System (VCS), like gitolite which is already used in the build pipe, is exposed to the plugins developers. Upon plugin reception, a hook in the VCS triggers a notification to the PVs through the `MQTT` broker. PVs then request the source code of the notified plugins via the PR gateway and later push back their verification results and STRs. Finally, a User Interface (UI) allows easing the

[8] `https://docs.docker.com/registry/`

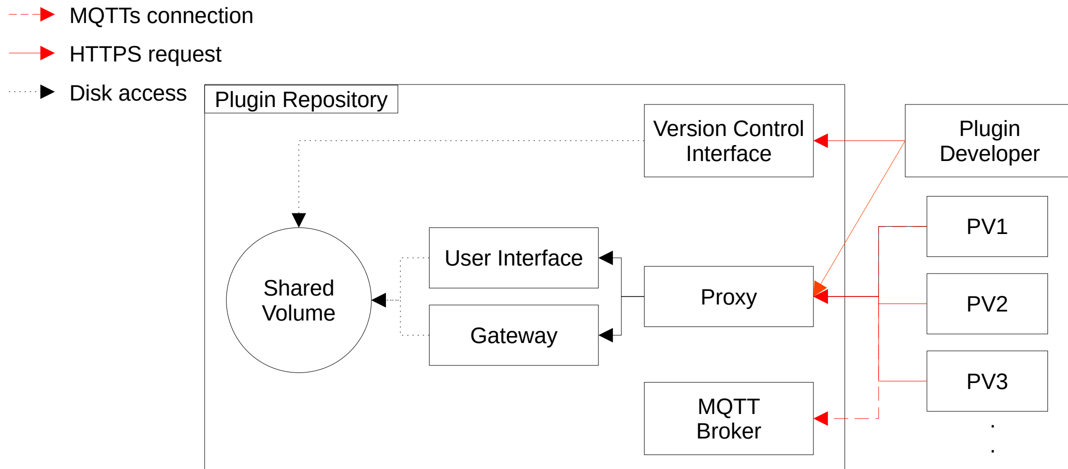


Figure 3.6: Full PR architecture.

interactions between the PR and the plugins developers. A proxy, like HAproxy, is added in front of the Gateway and the UI in order to multiplex those services is on the same port, namely the 443 one for `HTTPS` traffic.

`PQUIC` peers interactions Besides the verification goal of this distributed system, `PQUIC` peers are expected to interact with it. Namely, they must be able to pull plugins bytecode and Merkle tree's STRs. Currently, `PQUIC` is not modified for such interactions with the SPMS and this should be done to fully implement the SPMS.

Section 3.4.2 discusses the plugin validation process on the pluginized peer side as well as some timing considerations. It appears that caching the STR content could reduce the validation overhead. To this purpose, one could consider to include a validity tag in the STR, similarly to the ones embedded in SSL certificates. One could also consider returning a plugin binding and the path to reach it in the Merkle tree in a single request. This could limit the number of requests a pluginized peer should perform to validate the safety of a plugin.

Binding Manager The in-memory database presents two main drawbacks. First, the space issue, too many plugins might fill up the service memory. A longer term alternative would be to write the data on disk at periodic intervals. Second, the concurrent accesses protection is very basic and not optimized. Other protections like fine-grained locks, lock-free schemes with compare-and-swap operations or even the usage of full-fledge No-SQL databases (like Redis) are practicable to optimize this service for heavier traffic.

Binding Generator A custom format for the bindings is practicable, but this approach is error prone and there is no need to re-invent the wheel. It was the initial choice but it showed poor ability to scale with new requirements which motivated to switch to the more common `tar` format. That being said, it is important to notice that the additional metadata specific to the `tar` implementation will obviously change the hash of the binding. This implies an extra care in the choice of the tools to remain coherent in the whole system. We choose the `GNU` version of the `tar` format even if, in practice, the Python implementation which is used in this service showed some small differences with the CLI version. A more detailed survey of alternate binding formats could be interesting.

Build pipe The early versions of this component relied on Buildah[9] and Podman[10]. Those tools were chosen to build the SPMS micro-services containers within the controller service, which is itself a container. However, due to possible regressions in Buildah, some required images were not building anymore[11], so this solution was dropped. The current alternative is leaking the Docker socket within the controller to allow it to build containers. This is absolutely not secure and should be replaced by a better solution.

SPMS side-channels The `MQTT` usage for inter-component communication enables the creation of side-channels. We discuss here two examples thought during the SPMS prototype development.

In conjunction with the build pipe, one could imagine an automated, on-the-fly, update system for the SPMS components. The build pipe could send a notification to the PR when a developer pushes a micro-service update. The PR would then forward this notification to all PVs using this specific micro-service. Finally, each notified PV would pull the update and apply it on its live micro-services stack.

One could also envision telemetry for system monitoring. Administrators could collect usage data as regularly verified properties or the number of plugins handled by a given PV. This kind of usage data should be useful to deploy optimized PVs embedding multiple verifiers.

3.7 Conclusion

In this chapter, we discussed our prototype implementation of the Secure Plugin Management System (SPMS) for plugins verification and distribution. We first

[9] A tool allowing to build OCI container images daemonless. See `https://buildah.io/`.

[10] A daemonless container engine with the exact same syntax as Docker. See `https://podman.io/`.

[11] Among others, `ubuntu:12.04` required for the `T2` verifier.

covered high-level design choices which guided the whole SPMS architecture. The system's components were then detailed and some deployment specific considerations were presented. We discussed some points of the prototype evaluation. The chapter is finally closed by some future works to complete the system and enable full interactions between the SPMS, the plugins developers and the `PQUIC` peers.

Now that we covered in details this basic implementation of the SPMS, we will look at a new verifier we implemented by leveraging a state-of-the-art formal verification framework. Among all the properties and verification techniques we considered during this work, we chose to formally verify `PQUIC` plugins for possible unexpected and/or malicious modifications of their connection context. The next chapter is dedicated to this new verifier.

Chapter 4

Side effects property verification

As a reminder, `PQUIC` allows users to tune the protocol's default behavior by leveraging the plugins concept. Those are collections of code chunks, called *pluglets*, which are provided by the users to overwrite some part of the protocol. A pluglet receives as input a `QUIC` connection context represented by the `picoquic_cnx_t` structure. Currently, pluglets have read and write access to a certain subset of this context's fields without explicit restriction. Our interest is the **limitation of the side effects on the context provided to pluglets**. In other words, we want to formally prove that a given pluglet will only modify fields for which it has the right to do so.

This chapter provides a detailed description of the side effects verification we perform on the `PQUIC` plugins as well as the challenges we faced and the limitations they imply on the current state of our work.

We begin by explaining what code is required in order to verify user-provided properties with the SeaHorn framework and how it is produced. We then describe our modeling of `PQUIC` with its limitations, the assumptions we made and their justifications. Next, the testing methodology is described with, among others, our tuning of SeaHorn and how we ensured that our model is accurate. The results of our verification efforts are then presented. Finally, we describe some future works which could enable a full verification of `PQUIC` plugins.

All the source code mentioned in this chapter is released as an open-source project on GitHub[1].

[1] `https://github.com/nrybowski/pquic-formal-model`

4.1 SeaHorn and annotated code

The SeaHorn verification framework [8] is used to formally verify user-defined properties, also referenced as annotations in this document. In this mode[2], it takes as input annotated `C` code and performs static verification i.e., it does not execute the input code but applies reasoning on its (optimized) LLVM bitcode[3] form.

This section first describes (*i*) how SeaHorn allows users to define their own properties and (*ii*) the kind of outputs it produces. It then covers the application of SeaHorn on `PQUIC` pluglets by detailing (*i*) the kind of annotations required, (*ii*) a custom format we propose to express the specifications of the protoops and (*iii*) how the annotations are automatically generated on the basis of those high-level specifications.

4.1.1 SeaHorn and regular `C` code

Annotated code Listing 4.1 shows a very simple example of annotated `C` program where the properties to check are specified using the `sassert` function. The `extern` function `nd` at line 1 is understood by SeaHorn as a function returning a non-deterministic integer value which is then constrained at line 5 using the `assume` function. Those three elements, `sassert`, `assume` and *extern* functions are the basic building blocks we extensively use to verify pluglets.

```
1  extern int nd();
2
3  int main(void) {
4      int x = 2, y = 1, t = nd();
5      assume(t >= 0 && t<=10);
6
7      for (int i=0; i<t; i++) {
8          x *= 2;
9          y += 1;
10     }
11 #ifdef SUCCESS
12     sassert(x >= y); // should unsat (pass)
13 #else
14     sassert(x < y); // should sat (fail)
15 #endif
16     return 0;
17 }
```

Listing 4.1: Simple annotated program verified with SeaHorn.

[2] This framework is also able to perform other kind of verifications but we are mainly interested by the annotated property verification.

[3] Specific encoding of the LLVM Intermediate Representation (IR). See `https://www.llvm.org/docs/BitCodeFormat.html#abstract`.

SeaHorn outputs SeaHorn outputs `unsat` when none of the provided assertions (`sassert` calls) is violated and `sat` if at least one is. We thus define a *negative* result as receiving an `unsat` when the Code Under Analysis (CUA) respects all its specifications and a *positive* result when at least one does not. By extension, we define a *false negative* result when the Software Model Checker (SMC) outputs `unsat` rather than `sat` and inversely for a *false positive*. Those concepts of `[un]sat` are clearly illustrated in Listing 4.1. Line 13 defines a property which should always be true because `x` is growing faster than `y`. Hence, SeaHorn is expected to always return `unsat` in that case. Inversely, it should always output `sat` for the property at line 15.

An advanced feature of the SeaHorn verification framework is its ability to produce executable counter-examples (CEX) [28]. Those are execution traces of the CUA showing how to reach an error state from the initial state by attributing values to the variables leading to the error. The execution of such counter-example is expected to print the message `__VERIFIER_error was executed` which indicates that the counter-example effectively triggered the erroneous state. By extension, it implies that the error found by the solver really exists. Cases where the CEX is unable to reach the error are possible and indicate that we may not draw any conclusion on the result [7]. That being said, Gennari et al. [28] explain that generating such counter-examples is quite difficult especially when dealing with advanced memory usage. They propose some manual workarounds to circumvent the current CEX generation limitations.

During our experimentation, we encountered cases of "strong" false negatives corresponding to some SeaHorn unexpected outcomes. Those are probably due to too deep code optimizations before running the solver or because of bad pointer manipulation. Typically, we can test such behaviors by adding a final `sassert(false)` at the end of the annotated main. This assertion is expected to fail whatever code is provided previously and thus obtaining `unsat` shows that the verification output does not make sense. However, we noticed cases where some failing assertions were legitimately unreachable e.g., because of infinite loops (see Appendix A.3).

4.1.2 SeaHorn and `PQUIC`

Two problems arise when we consider code annotations: (*i*) which annotations to produce and (*ii*) how to generate them? The answer to these questions is dependent on the property we want to check. Our current interest in this chapter is the limitation of the side effects on the context provided to a pluglet. Indeed, each protoop is expected to present a well-defined behavior. This means that a pluglet implementing a given protoop is allowed to only modify some authorized fields of the context it receives.

Which annotations? We implemented a generic approach suitable for the verification of user-defined annotations which is similar to the one proposed by Zakharov et al. [50] for the static verification of operating system kernel modules. It requires two main elements: an *environment model* which abstracts the environment in which the Code Under Analysis (CUA) should run in real operation and an *API model* which describes the interactions between the CUA and its environment.

The verification method in itself is the following. First, a dummy context is initialized as well as the different inputs defined by the protoop specifications. Second, all the initialized data is cloned to generate a witness state before the CUA. This state should not be modified and is used to record the context before the CUA. Third, the dummy context is compared with the witness to verify which fields have been modified or not right after the CUA.

This method indicates which annotations are required. We need *helpers* to initialize a context, to clone it in a witness state and to assert that two contexts are the same except for some authorized fields. Such helpers are also required for each structure reachable through the context passed to the pluglets.

Annotated main The helpers by themselves are useless as SeaHorn requires to receive a function as input in order to perform its verification. We thus have to provide the annotated code, called *annotated main* in this document, implementing our testing method using the helpers we just defined. While SeaHorn is able to take any function as input, we provide a main in order to leverage the CEX inspection ability.

```
int main(void) {
    picoquic_cnx_t cnx, cnx0;
    init___picoquic_cnx_t(&cnx);
    cnx.protoop_inputc = 1;
    cp___picoquic_cnx_t(&cnx, &cnx0);

    uint64_t current_time = dummy___uint64_t();
    cnx.protoop_inputv[0] = current_time;

    set_next_wake_time(&cnx);  // Pluglet call
    assert___picoquic_cnx_t(&cnx, &cnx0, WAKE_NOW);

    return 0;
}
```

Listing 4.2: Simplified annotated main example.

Our proposed verification method is flexible enough to adapt to multiple use-cases. We consider each pluglet as a black box from which we want to observe the global impact on its input context. This means that the CUA is the entire

pluglet call on the dummy context as illustrated in Listing 4.2. Lines 2 to 8 show the preparation phase consisting of the initialization and duplication steps. Line 10 represents the CUA, i.e. the pluglet call. Finally, line 11 shows the comparison between the witness context and the one passed to the pluglet. It verifies that the whole context is unchanged except for the `WAKE_NOW` field. The `WAKE_NOW` key is illustrative, more complex access rules may be expressed as e.g., `WAKE_NOW | DATA_SENT`[4] and so on. A special `ASSERT_NONE` key is also defined to specify that no field is mutable.

Note that the return value of the pluglet call is ignored since it does not impact the side effects property. The set of helpers forms, in conjunction with the annotated main, our environment model which we simply call `PQUIC` model in the rest of this document.

High-level protoop specifications In order to generate such annotated main, we have to somehow know (*i*) what inputs are required for a given protoop and (*ii*) which fields are legitimately mutable. To this end, we propose a high-level representation of the protoop specifications which is illustrated in Listing 4.3.

```
{
    "name": "update_ack_delay",
    "inputs": {
        "pkt_ctx": ["picoquic_packet_context_t", "*"],
        "old_path": ["picoquic_path_t", "*"],
        "rtt_estimate": ["int64_t"],
        "first_estimate": ["bool"]
    },
    "outputs": {},
    "effects": {
        "pkt_ctx": ["ack_delay_local"]
    }
}
```

Listing 4.3: `update_ack_delay` protoop high-level specifications.

This format contains at least four fields: the name of the protoop, its inputs, its outputs and the allowed effects on the context. The inputs are denoted as arrays to indicate whether the variable's value or reference is stored into the context. Also, "inner effects", such as modifying a field in a sub-structure of the context, are currently neither represented in the specifications nor supported by our verification method.

How to generate the annotations? The `PQUIC` structures contain many fields and generating helpers to handle them by hand would be error prone and not flexible

[4] It indicates that both variables are allowed to be modified by the pluglet.

enough to adapt for future structures extensions. This motivates automatized helpers generation. The main method used here is meta programming based on Abstract Syntax Trees (AST) transformation i.e., a program manipulates another program encoded in an abstracted form.

Helpers Those functions are generated based on the header files. We designed a tool[5] performing the following tasks. First, a `C` AST is produced from the `PQUIC` header files using the `pycparser`[6] Python library but its representation is quite difficult to manipulate. Therefore, it is pre-processed to produce an easier to handle intermediate AST. It is represented as a hashmap whose keys are the structure names. The values are sub-hashmaps containing the field names and their types. Then, a new `C` AST is produced. This one contains the definition of the three helpers for each `PQUIC` structure. Finally, `C` code and associated header files are generated from the helper's AST. This transformation process is illustrated in Figure 4.1.

Figure 4.1: Helpers generation process.

In its current state, the helpers generator is not fine-grained in its output, i.e. it includes more fields than required. It thus requires some manual updates as removing fields or adding some constraints to reach the current model. For example, the `picoquic_packet_t` structure, which represents a packet, contains a `bytes` field. It represents the packet's payload under the form of an array of 1536 bytes. On one hand, handling explicitly that amount of data with SeaHorn slows down the verification process. On the other hand, this field is not yet modifiable by pluglets so it does not need to be verified. Hence, this field has been manually removed from the generated helpers.

Annotated main generation We then have to generate the annotated mains which will be provided to SeaHorn, one for each protoop. The protoop specifications, illustrated in Listing 4.3, are used to produce the AST of the main. First, the context is declared, initialized and duplicated by leveraging the helpers

[5] Available at `https://github.com/nrybowski/pquic-formal-model`
[6] `https://github.com/eliben/pycparser`

previously produced. The same sequence of operations is then applied for each input specified. Afterwards, the CUA is called and directly followed by context and *mutable* context's inputs comparison. Once again, the AST is used to generate the annotated main `C` code.

4.2 Methodology

This section is dedicated to the methodology we applied for the `PQUIC` model development and usage. It firstly details the exact SeaHorn framework tuning we used. It then describes the model testing method we created to evaluate the model accuracy and adapt its behavior if required. Finally, it covers the hardware and software platform we worked on.

4.2.1 SeaHorn tuning

SeaHorn is a tool which provides a vast amount of options to tune its behavior according to our requirements. Here is a summary of the non-default options we use with their justification:

bpf This is a shortcut[7] for the `clang|pp|ms|opt|unroll|cut-loops|opt|horn` command sequence which details how the `C` code is processed before being passed to the solver. The first stages, `clang` and `pp`, are the compilation and pre-processing phases in charge of converting the `C` code in LLVM bitcode and performing a first optimization pass. `ms` stands for mixed semantics transformation. The next four phases are a first LLVM compiler specific optimization, followed by loop unrolling and cutting then ended by a last LLVM specific optimization pass. Finally, the *Constrained Horn Clauses* (CHC) are generated, on basis of the optimized LLVM bitcode, and solved.

--bmc=mono --inline The `PQUIC` inner structures contain many bit fields which are required to be verified but the default CHC engine used by SeaHorn does not support bit-precise reasoning. We thus tune the CHC engine as well as the Bounded Model Checker (BMC) engine and explicitly ask for function inlining according to Gurfinkel [9, 17].

--track=mem This switch tunes the memory model. According to Gurfinkel et al.[30], this option allows modeling integer scalars, pointer addresses and the heap. The latter is represented as a collection of non-overlapping areas.

[7] While this naming is quite confusing, it has nothing to do with neither the `eBPF` assembly nor its Linux kernel implementation. It is a notation specific to SeaHorn.

--dsa=sea-cs This option allows SeaHorn to make usage of a new context-sensitive memory model for `C` programs verification by Gurfinkel and Navas [31]

--cpu=600 --mem=4000 During our initial tests we had cases of checks taking multiple hours ending up by consuming all the available memory. Such runs were then killed by the kernel to free up the space and clean up the process. Based on the successful runs we experienced, a time limit of 10 minutes and a space limit of 4 GB should be sufficient for our tests.

--cex=witness.ll --log=cex --bv-cex --horn-bv-part-mem These switches are used to produce executable counter-examples (CEX) with bit-level precision according to the SeaHorn's documentation [6].

4.2.2 Model testing

To limit the number of false negatives and false positives produced by our model, we leverage the CEX and `sassert(0)` testing methods previously introduced on specific custom pluglets.

False negative check In the case of false negatives, those are protoop's implementations we wrote to explicitly violate their high-level specifications. Listing 4.4 shows a minimal pluglet modifying one of its inputs while it is read-only according to its specifications. Figure 4.2 illustrates the logic we use to test our model for false negatives.

```
1 protoop_arg_t is_ack_needed(picoquic_cnx_t *cnx) {
2     picoquic_path_t* path_x = (picoquic_path_t*) get_cnx(cnx, AK_CNX_INPUT, 2);
3     set_path(path_x, AK_PATH_SEND_MTU, 0, 1);
4     return (protoop_arg_t) 0;
5 }
```

Listing 4.4: Illustration of minimal faulty pluglet for the `is_ack_needed` read-only protoop which modifies one of its input.

Main test branch We begin by a first execution of SeaHorn whose expected output is `sat`. If the result is neither `unsat` nor `sat`, something went wrong and the output is unknown. It is generally due to compilation errors at the very first stages of the SeaHorn pipe or because of timeout.

In case of `sat`, we produce and then execute the generated CEX, searching for the `__VERIFIER_error` message. If the error is successfully triggered, we ensure that the model under test produced a correct positive result. Otherwise, we cannot draw any conclusion on the output and this indicates that we have to refine our

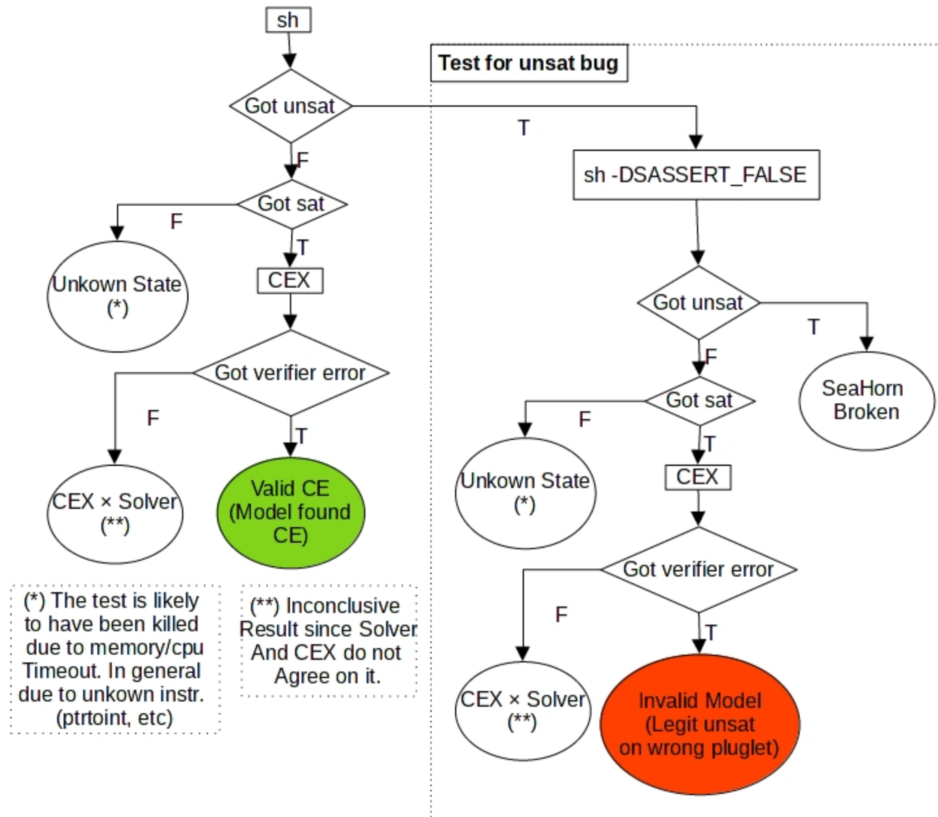


Figure 4.2: Model test procedure for false negatives. The green and red outcomes are inverted when considering false positive testing.

model, either because we fell in a pitfall of SeaHorn's CEX generation[8], or because the model is simply inaccurate.

Faulty `unsat` check If the initial SeaHorn execution outputs an unexpected `unsat`, we check if it is a legitimate one i.e., it is not a bug with SeaHorn. To test this possibility, we re-run the SMC with an additional `sassert(0)` at the end of our environment model to force a `sat` answer. The same testing logic as for the main branch of Figure 4.2 is applied. That is, we check if the SeaHorn output is `unsat` or `sat`. In the first case, it means that the SMC is somehow broken by the CUA. For the latter case, we test the CEX. If the error is successfully triggered,

[8] In some cases, it is possible to circumvent those pitfalls by constraining the model. Such modifications are discussed in the next sections.

this implies that our model produced a false negative on a trivial faulty pluglet. Hence, it is not accurate enough and requires some refinement.

False positive check Regarding false positive testing, we write very simple pluglets known to respect their specifications. The exact same test scheme is used, only the green and red outcomes are inverted since we are expecting legitimate `unsat` for such test cases.

We also emphasize that this testing method is used to **estimate the accuracy of our model on known code**. Using the tested model on real pluglets *might* trigger false negatives or positives, e.g. because the code under analysis is not well handled by SeaHorn or due to inaccuracy of our model. Interpreting the results of real pluglets verification is another task we discuss later.

Model test cases Table 4.1 details the 21 test cases on which the `PQUIC` model is run. Their source code is available under the `model_tests` folder of the model's repository[9]. Each test pluglet is named by the concatenation of its protoop name and a test case name. The latter briefly describes the structure which is modified. The special "*default*" test case name indicates that the test is named only according to its protoop because it does not modify any field and only gets the context's inputs. Such a case is mainly used to ensure the model behaves correctly on "empty" pluglet.

Testing environment All the experiments discussed in this document were run on the same machine, namely a laptop using an `Intel(R) Core(TM) i7-3610QM CPU @ 2.30GHz` (4 cores, 8 threads) CPU paired with 16 GB of DDR3 1600MHz RAM. It is running Arch Linux with kernel 5.11.8.

The `seahorn/seahorn-llvm10:nightly` docker container (hash `8dc104c2e93f`) is used for all the tests.

4.3 PQUIC modeling

It appeared, while generating the helpers and annotating main functions, that `PQUIC` is quite difficult to model for formal verification, at least with SeaHorn. We describe in this section the different issues we encountered and the limitations they imply on our current verification model.

The two first points focus on `PQUIC` related difficulties, i.e. its built-in memory separation and the diversity of its internal structures. Next, SeaHorn related

[9] `https://github.com/nrybowski/pquic-formal-model`

Type	Protoop	Test case	Field modified
False Negative	`congestion_algorithm_notify`	cnx[1]	`cnx->is_0rtt_accepted`
		cnx_if	`cnx->is_0rtt_accepted`
		cnx_inner[2]	`cnx->local_parameter.max_idle_timeout`
		in0[3]	`path_x->mtu_probe_sent`
	`is_ack_needed`[4]	cnx[1]	`cnx->is_0rtt_accepted`.
		cnx_inner[2]	`cnx->local_parameter.max_idle_timeout`
		in2[3]	`path_x->send_mtu`
	`update_ack_delay`	cnx[1]	`cnx->is_0rtt_accepted`
		cnx_inner[2]	`cnx->local_parameter.max_idle_timeout`
		in0[3]	`pkt_ctx->ack_needed`
		in0_inner[5]	`pkt_ctx->retransmit_newest->pkt_length`
		in0_sack_ll[6]	`start_range` of the latest `sack_item`
		in1[3]	`path_x->mtu_probe_sent`
		in2[7]	`cnx->protoop_inputv[2]`
		in3[7]	`cnx->protoop_inputv[3]`
False positive	`congestion_algorithm_notify`	*default*	/
		bit[3]	`path_x->bytes_in_transit`
		cwin[3]	`path_x->cwin`
	`is_ack_needed`[4]	*default*	/
	`update_ack_delay`	*default*	/
		pkt_ctx[3]	`pkt_ctx->ack_delay_local`

[1] Direct context's field access e.g., `set_cnx(cnx, AK_CNX_IS_0RTT_ACCEPTED, 0, 1)`.
[2] Inner context's structure field access e.g., `set_cnx(cnx, AK_CNX_LOCAL_PARAMETER, TRANSPORT_PARAMETER_MAX_IDLE_TIMEOUT, 1)`.
[3] Context's input direct access.
[4] Read-only protoop. For false positive check, no field is modified.
[5] Structure access through a pointer contained in a context's input.
[6] Linked list modification.
[7] The tested field should not be writable with the used syntax in real pluglets but the use-case is still tested.

Table 4.1: Summary of the model test cases.

limitations are detailed. They cover issues highlighted in minimal examples but having a direct impact on the `PQUIC` model use-cases coverage. The next point discusses semantic constraints, which points out the network protocol verification aspect. Finally, this section ends with a set of working hypothesis that we define to oversee the `PQUIC` model usage.

4.3.1 Memory separation

As explained in Section 2.1, `PQUIC` uses a strong memory separation between the core of the protocol and the Pluglet Runtime Environment (PRE) but SeaHorn is not aware of that. This implies that our `PQUIC` model should take into account this separation by design. Indeed, it is better to avoid useless and potentially complex verification of unreachable fields from within the `uBPF` VM.

In practice, we do not consider the fields which are not writable through setters of the PQUIC's API during the duplication and comparison phases. However, some unwritable fields are still initialized in order to avoid `Read of undefined values` warnings from SeaHorn which may lead to incorrect answers from the Software Model Checker (SMC).

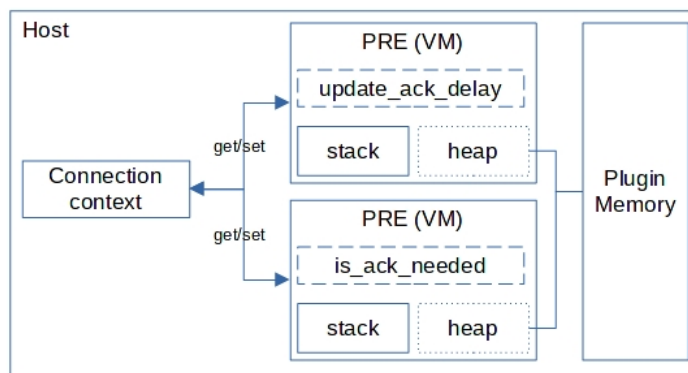


Figure 4.3: PQUIC memory separation illustrated for the `ack_delay` plugin.

It is legitimate to ignore the verification of fields that are not handled by the PQUIC API helpers. Indeed, the connection context provided to the pluglets lies in the host memory. Therefore, the pointer to the context and the ones that it contains are not directly reachable from within the PRE as illustrated in Figure 4.3, where the pluglet stack is entirely isolated from the host memory. In fact, the uBPF VM is expected to kill pluglets trying direct accesses to memory areas outside the VM. However, the `get/set` operations from the API are not executed within the VM and thus perform legal accesses in the host memory.

4.3.2 PQUIC structures

The `picoquic_cnx_t` structure and its inputs, which are passed as arguments to each pluglet, contain different kinds of variables: direct access fields, embedded structures, simple pointers toward a structure or a buffer, double pointers forming dynamic arrays of structures and, finally, arrays of native C types or PQUIC specific structures. Notice that some of those PQUIC structures, such as `picoquic_sack_item` or `picoquic_packet_t`, are also the building blocks of more complex data structures such as simple and double linked lists. We also have to consider `picoquic_metadata_t` which is the base block of hashmaps.

In order to illustrate how those kinds of variables may be “chained“, and thus produce very complex structures to verify, let us consider `picoquic_cnx_t`. It contains a double pointer `path` forming a dynamic sized array[10] of `picoquic_path_t` structures. Each of those paths contains a fixed size array of `picoquic_packet_context_t`. This structure mainly contains a simple linked list of `picoquic_sack_item_t` and two double linked lists of `picoquic_packet_t` (one for the packets to retransmit and the other for the already retransmitted ones) with pointers to the head and the tail as shown in Listing 4.5.

```
1 typedef struct st_picoquic_packet_context_t {
2     [...]
3     picoquic_sack_item_t first_sack_item;
4     picoquic_packet_t *retransmit_newest, *retransmit_oldest;
5     picoquic_packet_t *retransmitted_newest, *retransmitted_oldest;
6     [...]
7 } picoquic_packet_context_t;
```

Listing 4.5: Sample of `picoquic_packet_context_t` structure’s fields.

Obviously, each of those fields are mutable through the `get/set` helpers provided by the `PQUIC` API. We thus have to provide to the CUA, in theory, a model of the pluglet context which allows handling and verifying all those indirect fields. In practice, this revealed to be a bit complex as we will discuss in the next section.

4.3.3 Practical SeaHorn limitations

Pointers toward structures Manipulating pointers in SeaHorn is possible but is strongly tied to the memory model used by the SMC. The study of such models is beyond the scope of this document but could explain some limitations encountered during our `PQUIC` model development.

Simple pointers Such pointers may be initialized in three different ways: by using a non-deterministic function such as `extern my_struct *nd()`, by using `malloc()` calls to reserve some space on the heap or by referencing a variable on the main’s stack. The latter is achievable either by declaring a structure and initializing each of its fields explicitly or by calling some non-deterministic function such as `extern my_struct nd()`. We provide a detailed discussion of structure initialization with SeaHorn in Appendix A.1 but here are the main conclusions we drawn.

The solver outputs the expected result for each method, the issues encountered are only CEX related. Using a non-deterministic function returning a pointer toward a structure is not a suitable approach in our use case because this triggers

[10] The size is given in the `nb_paths` field.

segfaults during CEX executions. Relying on non-deterministic functions which return a structure is not sufficient because CEX is unable to reach the error state. This issue has been mainly noticed for CUA setting fields to 0 which seems to indicate that the CEX is initializing by default all the fields of the structure to 0. Explicitly setting each field with non-deterministic function calls solved this issue.

On basis of this knowledge, we use pointers toward both the stack and the heap, as illustrated in Listing 4.6[11]. They show in general good results i.e., the model is performing as expected.

```
picoquic_packet_t *p1;
p1 = (picoquic_packet_t*) malloc(sizeof(picoquic_packet_t));
init___picoquic_packet_t(p1);
param1->retransmit_newest = p1;
param1->retransmit_oldest = p1;
/* -- or -- */
picoquic_packet_t p;
init___picoquic_packet_t(&p);
```

Listing 4.6: Simple pointer initialization with SeaHorn

Nevertheless, using this method to verify the `send_path` pointer toward a `picoquic_path_t` in the `picoquic_packet_t` structure seems to somehow break the model. Indeed, calling the comparison helper on this specific field causes the model to timeout within a 10 minutes limit. By removing the verification instruction, we obtain an output from the SMC after around 2.5 minutes. By removing the duplication instruction, the SMC finishes after around 1 minute. Finally, by removing the initialization of the field, an output is received after 10 seconds.

This is limiting because on one hand we have a field, hiding many more, which is left unverified and on the other hand, it is not clear why this field in particular is problematic while others[12] are verified as expected. Especially, we do not know if the explicit initialization of the structure may introduce an overhead on the solver side which could explain such timeouts or if the memory model becomes too complex for the solver to handle.

Double pointers Even if we showed that SeaHorn is able to handle double pointers (see Appendix A.2), we were not able to manipulate a dynamically sized array of pointers toward structures within our `PQUIC` model. This leads to a strong limitation in the scope of verifiable protoops since we are not able to handle the `path` field of the pluglet context (`picoquic_cnx_t`).

[11] `init__picoquic_packet_t` is a function which explicitly initialize each field of the structure by calling non-deterministic functions.

[12] Such as `picoquic_packet_t` fields from `picoquic_packet_context_t`.

However, we were able to initialize such an array of paths in our `PQUIC` model with a single entry by hard-coding the size field to 1 as a workaround. This value was not chosen at random but is the result of some experiments. We tested different ways to express the size stored in the `nb_paths` field which is firstly assigned by a non-deterministic function call returning an integer.

We first explicitly constrained its value with `assume(p->nb_paths > 0)` or `assume(p->nb_paths < 2 && p->nb_paths > 0)`. Both constraints led to CPU timeouts on both false positive and false negative checks with limits of 10 and 30 minutes. Interestingly, only the `update_ack_delay_in2` and `update_ack_delay_in3` false negative test cases ended successfully within a few seconds. The reason why those specific cases pass is still unclear.

We then tested hard-coded values, even if we know by the experiences described in Appendix A.2 that they show worse results in terms of false positive outputs. Table 4.2 summarizes the values experimented and the model behavior. While the outcome observed when `nb_paths=2` corresponds to the one highlighted by our minimal example, the others were not expected. Since a size of 1 successfully passed our benchmark, it is integrated in our `PQUIC` model. Currently, it is not clear if the issue is related to the memory model used by SeaHorn, if it is due to the optimizations performed on the loops used to initialize the array or if it is linked to the pointer accesses.

`nb_paths`	Model benchmark result
10	All the test cases ended with a buggy[1]`unsat` outcome.
2	False negatives: all the instances passed. False positives: all the instances failed with buggy[1]`unsat`.
1	All the test cases succeeded.

[1] `unsat` on explicit `sassert(0)`.

Table 4.2: Double pointer hard-coded size experiment results.

Table 4.3 shows an exhaustive list of directly impacted pluglets where the problematic field is accessed via a call to `get_cnx(cnx, AK_CNX_PATH, i)`. Those pluglets are thus susceptible to either need to access more than one path or to modify the paths they get. Due to the limitations we face, our model is only able to verify the side effects of pluglets for exactly one path. This is obviously a strong limitation for the `multipath` plugin but also, in general, for a complete side effects restriction check.

Linked data structures Linked data types are obviously composed of structures containing pointers towards nodes of the same nature. From the previous points, we learned how to initialize a structure such that both the SeaHorn's solver and

plugin	protoop	hook[1]	pluglet[2]
no_pacing	set_next_wake_time	R	set_next_wake_time_without_pacing
qlog	connection_state_changed	PO	cnx_state_changed
multipath*	set_next_wake_time	R	X
	process_possible_ack_of_ack_frame	R	X
	write_frame	R	write_add_address_frame
	get_incoming_path	R	X
	parse_frame	R	parse_add_address_frame
	schedule_path	R	schedule_path_rtt
		R	schedule_path_rr
	schedule_frames_on_path	R	schedule_frames
	get_destination_connection_id	R	X
	predict_packet_header_length	R	X
fec*	/	/	finalize_and_protect_packet
datagram*	/	/	/
tlp	set_next_wake_time	R	X
	retransmit_needed	R	X
basic	set_next_wake_time	R	X
	retransmit_needed	R	X
	schedule_frames_on_path	R	X

* Calls `get_cnx(cnx, AK_CNX_PATH, i)` inside the plugin's helpers (Generally in a file called `bpf.h`). If no protoop is mentioned, it means that the accesses to the `path` field are limited to the plugin's helpers.

[1] R: replace, PO: post.

[2] If the pluglet has the same name as its protoop, this field is marked with an X for readability purpose.

Table 4.3: Summary of the protoops and their implementation affected by the double pointer issue. The table is sorted by plugin.

the CEX execution agree on the outcome of a CUA which uses a pointer to modify some of the structure's fields. We thus leveraged those techniques to extend the `PQUIC` model in order to verify the limitation of side effects on linked lists reachable through the pluglet's context. However, we again faced some unexpected behaviors from the SMC.

Simple linked list The main simple linked list of a pluglet context is the one formed by `picoquic_sack_item_t` nodes in the `picoquic_packet_context_t` structure. Such list is hard-coded in our model with at least one node and is then chained with a non-deterministic number of following nodes. At first sight, no limitation is observed on this one, the initialization, duplication and comparison helpers are working as expected and pass the model testing benchmark.

However, when we try to constrain the number of subsequent nodes to be at least 1 with an `assume(sack_ll_size > 0)`, the benchmark output is a bit different. All the test cases related to the `update_ack_delay` protoop, both for false positive and negative checks, are failing with `unsat` on an explicit `sassert(0)` call. While

the issue in itself is unexpected, it makes sense that issues are only triggered by this protoop since it is the only one, among the ones experimented, handling a `picoquic_packet_context_t` as one of its inputs. Contrary to the previous double pointer experiment, hard-coding the list size to 10 or even 1 did not change the outcome of the benchmark which is still outputting `unsat` on explicit `sassert(0)` call. Also, we may not add simple test cases with hard-coded list accesses such as `head->next->item = new_value` since a very simple counter-example for that is a list containing only one element. Because of those issues, it is unsure that cases where the linked list contains more than one element are really considered during the verification.

Doubly linked list Even with the limitation observed on simple linked list, we tried to verify doubly linked lists of `picoquic_packet_t` which are stored in the `picoquic_packet_context_t` structure. Hard-coding the initialization, duplication and comparison of a list containing a single element successfully passed the model benchmark.

A first step to verify lists with more than one element is to replace the verification code with an iteration on all the nodes, all else being equal. This is performed in the exact same manner as for the simple linked list check. Interestingly, it results in some test cases failures. Both false positive checks on the `update_ack_delay` pluglet fail with a `sat` outcome whose CEX execution is unable to reach the error location. The same behavior is observed for the false negative `update_ack_delay_cnx_inner` test case which experiments with inner context structure modification.

While the origin of such outcomes is not obvious, we also replaced the duplication code with a loop. This produced exactly the same outcomes as previously.

As a last step, the initialization code is replaced by a loop which produces a non-deterministically sized bounded list. In addition to the previous failures, the `update_ack_delay_in0_inner` false negative test case also fails with a CEX not reaching the error location.

Due to those unexpected outcomes, we stick with a single node hard-coded list.

Hashmap One feature of the Plugin Runtime Environment from `PQUIC` is to provide a way for pluglets of a given plugin to share data between them. This is achieved by attributing a specific identifier to a given area in the plugin's memory in order to retrieve it consistently between its pluglets. Under the hoods, this feature is enabled by the `plugin_struct_metadata_t` structure which forms the building block of a hashmap.

This metadata map is integrated in multiple major structures of `PQUIC` such as the context, path, packet and packet context structures. Helpers, such as `get_cnx_metadata`, are provided through the `PQUIC` API to allow pluglets to

handle those memory areas. They typically reserve some space in the plugin's memory if no metadata is already associated to the calling plugin during the `get` operation. In the end, this corresponds to a modification in the plugin context.

While those helpers are not directly called within pluglets, they are typically part of `get` functions defined in specific plugins. This is illustrated by the `get_westwood_state` helper, shown in Listing 4.7, which is only part of the `be.michelfra.westwood` plugin. It clearly shows that when the identifier does not refer to an already allocated memory area, it is initialized and stored in the metadata (lines 2 to 5 of Listing 4.7). It results in a second plugin context modification. This indicates the importance to check upon pluglets calls that only authorized data in those memory areas are written.

```
1 westwood_state_t *state_ptr = (westwood_state_t *) get_cnx_metadata(cnx,
      WESTWOOD_OPAQUE_ID);
2 if (!state_ptr) {
3     state_ptr = initialize_westwood_state_t(cnx, current_time);
4     set_cnx_metadata(cnx, WESTWOOD_OPAQUE_ID, (protoop_arg_t) state_ptr);
5 }
6 return state_ptr;
```

Listing 4.7: Content of a helper of the Westwood Congestion Control plugin getting the current westwood state stored in metadata.

That being said, it appears that the hashmap implementation[13] used by `PQUIC` is quite difficult to handle with SeaHorn which produces erroneous outputs even for minimal examples (discussed in Appendix A.4). This means that all the data and operations relying on the metadata feature, as well as their impact on the context provided to pluglets, are currently unverifiable.

4.3.4 Semantic constraints

So far, we considered the `PQUIC` structures only as a bunch of fields without paying attention to their signification. That being said, we are verifying parts of a network protocol and there exists a relation between those fields. We could illustrate that with the simple `picoquic_sack_item_t` structure, the start of the sack range should not be greater than its end. This requires an extra constraint to add in the initialization code such as `assume(start_of_sack_range <= end_of_sack_range)` to ensure that SeaHorn will not produce a CEX when it is not the case. Another, more complex, example of such fields relation is found in the `picoquic_path_t` structure which contains two `struct sockaddr_storage` containers for the local and peer addresses data and their length fields (`{peer, local}_addr_len`). It would not make sense to store the size of an `IPv6` address when the peers are

[13] `https://github.com/troydhanson/uthash`

operating over `IPv4`. We should then add some constraint as the one in Listing 4.8 but it seems to be difficult for SeaHorn to apply them because it made our model timeout on the 10 minutes time limit.

```
assume(
    (path->local_addr.ss_family == path->peer_addr.ss_family) &&
    (path->local_addr_len == path->peer_addr_len) &&
    ((path->local_addr.ss_family == AF_INET &&
        path->local_addr_len == sizeof(struct sockaddr_in)) ||
    (path->local_addr.ss_family == AF_INET6 &&
        path->local_addr_len == sizeof(struct sockaddr_in6))
);
```

Listing 4.8: Example of semantic constraint on `IP` address size related to its type.

Another kind of semantic constraint we have to add is the case of inaccurately typed fields. For example in the `picoquic_cnx_t` structure, the size of the dynamic `picoquic_path_t` array (`nb_paths`) is encoded in a `signed integer` which requires the addition of a constraint `assume(nb_paths >= 0)`. Likewise for the number of inputs and outputs fields (`protoop_inputc`, `protoop_outputc_callee`). This kind of additional constraints could be avoided by using `unsigned integer` in this specific case but, in general, an extra care should be taken for the choice of the field types in the pluginized protocol structures to ease the work of the SMC.

4.3.5 Working hypothesis

As we discussed in the previous points, we have to strongly limit our current `PQUIC` model in order to make it usable with SeaHorn. Here is the exhaustive list of the working hypothesis we used to reach its current state.

Hypothesis 1. To ensure that a plugin respects the side effect property, each of its pluglets is separately verified without making any link between them.

This implies that the effects of a chain of pluglets is not verified. We thus do not consider cases such as a pluglet hooked at the `pre` anchor writing some data in the context metadata field for a pluglet at the `replace` anchor. The `PQUIC` model should handle such cases by providing a freshly initialized structure to the pluglet at the `replace` anchor.

Hypothesis 2. Direct access to a memory area external to the `uBPF` VM from within a pluglet fails and stops the execution of the pluglet. Solely external memory accesses through the `get/set` helpers provided by the `PQUIC`'s API are legal.

This hypothesis is the main one used for fields discrimination in term of verification. Full details about which structures and/or structure's fields are

considered in our model are provided in Appendix B with a justification based on the hypotheses defined here.

Corollary 2.1. If a `set` function exists but its implementation is not provided, the associated structure *might* be considered as unreachable.

There exist `set` helpers currently listing some fields as valid access keys but with no implementation of the expected behavior, we thus took the freedom of including or not such fields in our current model.

Hypothesis 3. Access to any internal `PQUIC` data is impossible through the memory manipulation helpers provided by `PQUIC`.

While hypothesis 2 is a good representation of the reality, we were able to show that hypothesis 3 does not hold. Indeed, it is possible to have full read and write access to any field reachable through the context by manipulating their pointer with the memory helpers. Though, we still consider it to define the fields having to be checked with our helpers since this is an issue that needs to be fixed into `PQUIC`. More details about this exploit are available in Appendix C.

Hypothesis 4. The API implementation is correct.

Since SeaHorn does not make assumptions on the behavior of functions for which it does not have the implementation, we had to provide the `get/set` helpers code to SeaHorn. This implies that an error in the API's implementation will be seen as an error in a pluglet during the verification made by SeaHorn.

A better approach we envision for future work is using the formal specification of those helpers within our annotated code. This way, we decouple the verification of the API from the one of the pluglet code. That being said, since explicit constraints showed verification duration extension, the scalability of this method in practice has still to be studied.

Hypothesis 5. The metadata fields are part of the `PQUIC` memory model which is different from the model used for formal verification.

Even if the metadata are stored in the plugin context, we do not consider their modification as side effects **on the context**. We consider that they are rather part of the plugin specific memory areas handling which is another property. This allows us to ignore the hashmap manipulation issue we noticed earlier.

4.4 Experimentation results

Among the 80 protocol operations defined in PQUIC, about 45 are effectively implemented in existing plugins. We verify 16 of those pluglets[14] with the PQUIC model under the working hypotheses and limitations discussed earlier. This results in 11 false negative outcomes and 5 successful verification[15].

Successful verification From the five successful runs, four of them represent three fully verified plugins. Each of those pluglets are part of the model benchmark so we consider such outcomes as reliable.

The last pluglet (retransmit_needed_by_packet) is an isolated pluglet from a fourth plugin (basic). Its protoop is not part of the model benchmark which may leave some doubts on this result. That being said, its successful verification may be explained by the fact that the pluglet does not contain any other function call than PQUIC API helpers.

Here are some details for the fully verified plugins.

be.mpiraux.ack_delay It is composed of two pluglets in replace mode for the is_ack_needed and update_ack_delay protoops. It does not use plugin specific helpers.

be.michelfra.disable_cc This plugin is composed of a single pluglet in replace mode for the congestion_algorithm_notify protoop. Its implementation is rather basic as it only sets the cwin field of the path_x context's input. Hence, the verification is performed without any additional modification.

be.michelfra.westwood As the previous plugin, it is only composed of a single pluglet in replace mode for the congestion_algorithm_notify protoop but its verification is somewhat trickier due to its behavior.

First, it relies on a data structure stored in the metadata of the context. It implies read and write accesses on the context metadata as well as read and write access to the data in itself. It is stored on the plugin heap and thus in the plugin context. As mentioned in Section 4.3, such accesses are currently not verifiable. Nevertheless, this issue is taken into account by Hypothesis 5.

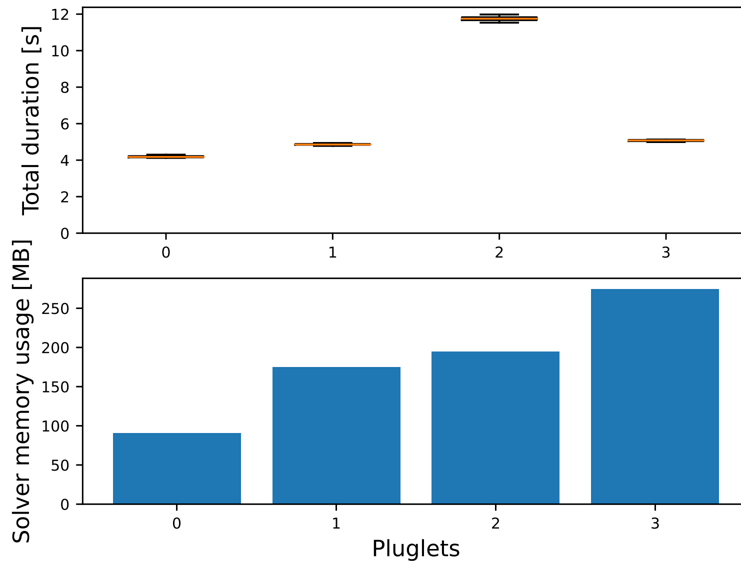
Second, writing such a structure in our dummy context is also difficult since it would require to add a new entry in the underlying hashmap which is not handled by SeaHorn. This is solved by hijacking the helper used to get

[14] They are chosen according to their input types. Some protoop are passing pointers toward buffers and this is not yet supported by the PQUIC model.

[15] The SMC returns unsat when run on the raw pluglet and sat when a sassert(0) is explicitly added.

the external data structure. Rather than accessing the context metadata, it returns a pointer toward a memory area freshly allocated and initialized with our verification helpers. This method allows to verify the pluglet impact on its context within the scope of our working hypotheses by bypassing the metadata accesses.

From a pure networking viewpoint, it should also be noticed that other congestion control schemes such as New Reno or Vegas use the same variables as the ones handled in Westwood, only the computations are different. This means that the modified context fields are the same as in this plugin. Therefore, we could expect that verifying another congestion control algorithm would produce the same result as verifying Westwood.



1 `be.michelfra.disable_cc/congestion_algorithm_notify` (12 SLOC)

2 `be.michelfra.westwood/congestion_algorithm_notify` (122 SLOC)

3 `be.mpiraux.ack_delay/udpate_ack_delay/` (21 SLOC)

4 `be.mpiraux.ack_delay/is_ack_needed` (27 SLOC)

Figure 4.4: Performance measurements of pluglets verification with the `PQUIC` model. Each experiment has been run 10 times.

Figure 4.4 shows performance measurement results from their verification. It shows for each of them (i) the total duration of the SeaHorn command pipe,

which includes the compilation of each source code file and (*ii*) the total memory consumption of the solver pass[16] only.

From the duration viewpoint, we notice that they are quite short, around some seconds, and very stable. This confirms that our 10 minute time limit is large enough to perform successful verification. Cases where this timeout is reached clearly indicate some issues either with the model or the CUA. The 12 seconds peak of the `update_ack_delay` pluglet may be explained by the fact that it is the only one taking a structure containing a non-deterministically bounded linked list.

About the memory consumption of the solver, it is quite hard to explain this rise without further knowledge of its internal behavior. We may still notice it requires a non negligible amount of memory ranging from 100 to more than 250 MB. This indicates that our 4 GB limit should be large enough for the kind of code we verify.

Unsuccessful verification Table 4.4 summarizes the additional pluglets we tested. They all failed with an `unsat` outcome on explicit `sassert(0)`. However, they also all showed many warnings[17] which reveal some badly handled pointers. They indeed use some plugin specific helpers handling the plugin metadata, such as explained for the `be.michelfra.westwood` plugin. They also call other protoops and / or external functions such as `setsockopts`. Those warnings are the reason of the unexpected verification outcomes. Applying the same hijack method on their helpers could solve this issue and produce the expected output. That being said, this experiment clearly shows that unaltered plugins might be tricky to verify with our model because of their heavy metadata usage.

4.5 Future work

While the current state of our formal verification of the side effects property is showing quite encouraging results, there is still some room for improvement. Here are some points we envisioned during our work.

4.5.1 Current `PQUIC` verification improvements

Annotated pluglets Verifying the global behavior of a pluglet is a good first step but it is not sufficient. Indeed, as we are verifying the pluglet as a black box, some internal malicious updates might happen without being externally seen. An example of such behavior is a pluglet locally saving a field of the context on its

[16] The last step of the command pipe.

[17] `detected ptrtoint instruction that might be escaping the analysis` and `inttoptr @ addr [...] is (unsoundly) assumed to point to a fresh memory region`

protoop	plugin
`update_rtt`	multipath basic
`schedule_frames_on_path`	multipath basic
`set_next_wake_time`	no_pacing multipath basic
`retransmit_needed_by_packet`	tlp
`header_parsed`	ecn qlog monitoring

Table 4.4: Additional pluglets verified.

own stack, modifying it in the context, calling a `PQUIC` helper making usage of the modified field or worse, another pluglet, then restoring this field. From the external point of view we consider, this behavior would not be detected even if it might deeply change the pluglet behavior.

This kind of situation implies that we also have to annotate the pluglet code in order to check, instruction per instruction, its behavior. It might be realized by applying our verification method for each instruction of the pluglet[18]. From a technical viewpoint, this should be feasible by producing a new, automatically annotated, pluglet code using AST transformation as experimented for the verification helpers and annotated main functions.

Invalid direct memory accesses As stated in Hypothesis 2, a direct memory access such as `cnx->nb_paths` *should* fail at runtime but SeaHorn will consider such instruction as valid. This means that within our `PQUIC` model, the side effect property will be verified even if this kind of direct access is wrong. One could consider a “malicious“ pluglet leveraging this “black-hole“ to make dummy usage of system resources by performing compute intensive tasks ending with such invalid memory access. Hence, this property should also be verified.

`PQUIC` memory usage property The side effect property, such as we defined it, ignores many of the advanced features enabled by the PRE [25] even if, in the end, they are also side effects of the plugin execution. Features related to memory usage such as plugin specific memory manipulation through `my_malloc` and similar

[18] One could also consider to only check the API calls. The advantages and drawbacks of both methods still require some thoughts.

helpers, pluglet shared data handling (ignored due to the limitation explained in Section 4.3.3) or QUIC frames reservation should also be part of an extensive verification.

That being said, some open question remain on how to handle that. While the ability of reserving new frames should be defined by the protoop specifications, the storage of data in plugin memory for future accesses is undefined as it is unknown in advance. One may consider a plugin whose pluglets are filling their dedicated memory with useless data on purpose of wasting host resources, even if the PRE ensures that it won't entirely fill up the host memory. Multiplying such plugins on many connections could lead in the worst case on a DoS of the host system. But how to verify automatically that a plugin presents such behavior?

Fields verification extension In Section 4.3.5 we defined our working hypothesis resulting from the practical limitations and modeling issues we encountered. By solving those issue, one could extend the side effect property verification to additional fields currently left untested.

In the same idea, we explain in Section 4.1.2 that currently, effects on inner structures are not handled by the PQUIC model. One could inspire from the PQUIC API helpers which specify an additional key indicating if an inner structure is queried as illustrated by Listing 4.9. This concept could be reused to extend the model's comparison helpers.

```
1 protoop_arg_t get_cnx(picoquic_cnx_t *cnx, access_key_t ak, uint16_t param);
```

Listing 4.9: `get_cnx` signature. The access key specifies the `cnx` field and `param` the field of the `ak` structure queried.

PQUIC's API formal model As stated in Hypothesis 4, we suppose the implementation of the PQUIC API is correct since we use it directly within our verification for simplicity. This may result in wrong verification result since a bug within the API will be seen as part of the Code Under Analysis (CUA).

In order to decouple the verification of the API from the pluglet's one, we envision specifying formally the API functions i.e., by asserting their preconditions and assuming their post-conditions. Moreover, this formal model of the PQUIC API could also be used to formally verify its correctness.

Simplifying VM side verification De Coninck et al. [25] state that multiple "simple" verification, such as exit instruction existence, valid opcodes and valid jumps verification, etc; are already implemented in their PRE. This implies such checks are executed at each plugin injection in the same fashion as the current eBPF kernel verifier.

To support the effort of offloading safety property checks of protocol plugins in an offline system allowing “once-for-all“ trust proof, one could envision to displace such verification from the PRE to the Secure Plugin Management System (SPMS)[25].

Model validation through other verification frameworks With the will to vary the verification results, we looked for other model checkers which might be suitable for our use-case.

We tested `ESBMC` [24] against a simple example of loop bounded by a non-deterministic value since this tool is using a syntax very close to the SeaHorn one. Unfortunately, we were not able to obtain a finishing execution. Because the non-deterministically bounded loop feature is crucial for our `PQUIC` model, we didn't look further with this tool.

We gave a try to the `Divine` model checker [18] but we encountered issues during the `PQUIC` source code compilation. Due to time constraints, we left this tool.

Because working with SeaHorn took some efforts, we didn't dig any further to find others frameworks but it could definitively worth it to extend `PQUIC` plugins formal verification with other frameworks.

4.5.2 Pluginized protocols design improvements

From our experience of trying to formally verify a specific protocol plugins property, we managed on one hand to observe some limitations with the tool we used but on the other hand, we also noticed some potential sources of improvement in the design in itself of future pluginized protocols.

Host memory misuse To solve the host memory manipulation hack through `PQUIC` API memory helpers we highlighted, one may consider adding some checks on the source and destination addresses in the `my_memcpy` and `my_memset` helpers to ensure the host's memory addresses are not hijacked.

That being said, it is currently unclear if such solution is applicable with static verification since the host memory addresses might be not correctly modeled, it might thus require runtime verification. For this purpose, we may envision a modified PRE with extra instructions for verification only and use a classical PRE for real operation. The main issue of this solution is that we loose the mathematical proof of formal verification we are looking for.

Field types semantic As seen in Section 4.3.4, inaccurate field types may introduce additional explicit constraints on the verifier side and thus add to the

verification complexity. It might be desirable to choose such types with an extra care in order to let the SMC handle this kind of details by itself.

Simpler context structure In order to reduce the number of hypotheses made for formal verification, and thus to deepen the accuracy of the `PQUIC` model by sticking closer to reality, one could envision to pass a lightweight version of the `PQUIC` structure to the pluglets. It makes even more sense from a security standpoint by reducing the attack surface reachable from the plugins, as some fields are clearly not expected to be usable by pluglets. Avoiding the exposition of such kind of fields to user-defined code may provide a “passive“ protection against possible unknown bugs in the pluginized protocol API.

While reducing the size of structures is doable in multiple ways, one could inspire from the `eBPF` Linux kernel implementation which uses an interesting technique to abstract kernel structures for users. A structure gathering the main fields of the kernel one is exposed to developers through the User API (UAPI). It is then used by the verifier to rewrite the memory accesses in order to let the user-defined BPF code directly interact with the kernel data in a controlled manner.

```
1 struct bpf_sock {
2     [...]
3     __u32 mark;
4     [...]
5 }
```

Listing 4.10: Sample of the `sock` abstraction in `eBPF` UAPI[19].

```
1 case offsetof(struct bpf_sock, mark):
2     BUILD_BUG_ON(sizeof_field(struct sock, sk_mark) != 4);
3     if (type == BPF_WRITE)
4         *insn++ = BPF_STX_MEM(BPF_W, si->dst_reg, si->src_reg,
5             offsetof(struct sock, sk_mark));
6     else
7         *insn++ = BPF_LDX_MEM(BPF_W, si->dst_reg, si->src_reg,
8         offsetof(struct sock, sk_mark));
9     break;
```

Listing 4.11: Instructions rewrite sample on the verifier side for the `sock` structure[21].

Listing 4.10 illustrates such abstraction for the `struct sock` kernel structure[22]. The verifier `convert_ctx_accesses` function[23] is called during the bytecode veri-

[19] https://elixir.bootlin.com/linux/v5.12/source/tools/include/uapi/linux/bpf.h#L4353
[21] https://elixir.bootlin.com/linux/v5.12/source/net/core/filter.c#L8824
[22] https://elixir.bootlin.com/linux/v5.12/source/include/net/sock.h#L451
[23] https://elixir.bootlin.com/linux/v5.12/source/kernel/bpf/verifier.c#L11213

fication process which, in turns, calls the appropriate bytecode rewriting routine (cfr. Listing 4.11).

However, the extensive approach illustrated in Listing 4.11 is doomed to disappear according to `eBPF` kernel implementation maintainers [2] due to scaling issues. It is expected to be replaced by using a newer concept called *BPF Type Format* (BTF) which describes data types and function information in a custom metadata format [1]. Nevertheless, the main idea remains interesting to move toward lighter structures exposed to pluginized protocols PREs.

Another direction already explored in `xBGP` [49] is the fully abstracted context i.e., no structure is explicitly passed to pluglets but they rather access underlying structures through API helpers only.

4.6 Conclusion

In this chapter we explained in detail the formal verification of the side effect property we performed on `PQUIC` plugins.

Especially, we described a `PQUIC` model suitable for such task as well as its limitations due to operational issues with the SeaHorn verification framework. We detailed how this model is automatically generated based on the `PQUIC` header files and a custom protoop specification format we proposed.

We also covered a testing method we implemented to ensure our model does not produce false negatives or false positives on simple faulty pluglets. Subsequently, 16 `PQUIC` pluglets out of the 45 existing ones have been verified. 5 of them were successfully checked while the others all failed due to pointer manipulation issues coming from the pluglet helpers.

Finally, we proposed some future works to enable full pluglet formal verification.

Chapter 5

Conclusion

Throughout this thesis, we explored some possible ways to distribute pluginized protocol plugins whose specific safety properties are verified. This work mixes several cutting-edge researches in both networking and software verification fields. We especially focused on the `PQUIC` protocol.

First, we implemented a prototype of the distributed *Secure Plugin Management System* (SPMS) proposed by the `PQUIC` authors. It includes a full design of the *Plugin Validator* component and a partial design of the *Plugin Repository* component. The security of the communications between all the SPMS components has been a strong focus of this work. The proposed designs are flexible enough to allow easy extensions for future use-cases.

Second, we leveraged a state-of-the-art formal verification tool, called SeaHorn, to verify a new property on `PQUIC` plugins. We were able to prove for three of them that they respect their specifications in term of write accesses on their connection context. Notably, one of them implements the Westwood congestion control algorithm for `QUIC`. We also showed through performance measurements that on-the-fly verification of plugins right before initiating a new `PQUIC` connection is not realistic. Hence, this supports the need of offline verification with specific environments such as the SPMS.

While this work shows encouraging results, the subject is far from being closed. We proposed numerous possible future works for both our contributions. Especially for the plugins safety proof topic, we explored the verification of a single property with a single method. In fact, there exist a plethora of other properties and methods to explore. The open-source nature of our contributions should ease their extension.

Interestingly, verifying protocol plugins shows similarities with verifying `eBPF` programs to inject in the Linux kernel. One could extend the SPMS principle to

support and/or replace the current in-kernel verifier. This could extend the number of networking use-cases made possible by the integration of a such generic in-kernel virtual machine. In general, there is a current trend of extending network protocols with user-defined code through pluginized protocols or the kernel `eBPF` VM. This shows the need for efficient code verification and distribution among an important number of end points. We hope this work is a little stone to this edifice.

Appendix A

Minimal examples of verification using SeaHorn

All the code chunks presented in this Appendix are minimalist Proof Of Concept used to simulate on a smaller scale the kind of annotated code required to verify the PQUIC structures accessible to pluglets. They are part of our PQUIC formal model repository[1].

The testing environment is set up by launching ./simple_wrapper example in the root of the repository. This drops the user in a shell whose current directory is /tmp of the SeaHorn container and where the host example directory is mounted at /mount. We used three commands to run those codes:

- gcc [<flags>] /mount/<C file> -o main && valgrind ./main Visually verify the correctness of the sample with simple printf calls. Valgrind is used to ensure the memory is correctly handled. In the positive case, memory usage is not expected to skew the experimentation with SeaHorn.

- sea bpf --cpu=300 --mem=1000 -m64 -g --bmc=mono --inline --track=mem --dsa=sea-cs --bv-cex --horn-bv-part-mem /mount/<C file> [<flags>] Run SeaHorn against the code sample and generate a witness file if the outcome is sat. Detailed explanation of those switches is provided in Section 4.2.

- sea exe -g -m64 --alloc-mem -o out /mount/<C file> witness.ll [<flags>] && ./out Link the witness bitcode with the sample code to generate a CEX and execute it.

For each minimal example we present the flags defined as well as their usage and we then discuss the experiments results.

[1] https://github.com/nrybowski/pquic-formal-model

A.1 Structure initialization

Since this is the basis of our verification method, we have to ensure the initialization of a structure is accurate enough to handle a large variety of CUA with SeaHorn. This example experiments with structure initialization through `extern` function call with SeaHorn, which corresponds to implicit structure initialization.

Listing A.1 defines the structures we will handle as well as their non-deterministic function definitions while Listing A.2 shows the main passed to SeaHorn. Finally, Listing A.3 presents the helpers implementation.

```
typedef struct {
    uint8_t uint2 : 1;
    uint8_t uint3 : 2;
    uint8_t uint1;
} my_struct;

typedef struct {
    uint8_t uint1;
    my_struct st;
#ifdef PTR
    my_struct *pt;
#endif
} my_struct2;

extern my_struct sh_my_struct();
extern my_struct2 sh_my_struct2();
```

Listing A.1: Test structures

```
int main(void) {
    my_struct2 s = sh_my_struct2(), s0;
#ifdef PTR_INIT
    my_struct i = sh_my_struct();
    s.pt = &i;
#endif
    dup2(&s, &s0);

    // Replace with CUA call

    cmp2(&s, &s0);
#ifdef FAIL
    sassert(0);
#endif
    return 0;
}
```

Listing A.2: Test case

```
inline void dup(my_struct *src, my_struct *dst) {
    dst->uint1 = src->uint1;
    dst->uint2 = src->uint2;
    dst->uint3 = src->uint3;
}

inline void dup2(my_struct2 *src, my_struct2 *dst) {
    dst->uint1 = src->uint1;
    dup(&src->st, &dst->st);
#ifdef PTR
    dst->pt = (my_struct*) malloc(sizeof(my_struct));
    dup(src->pt, dst->pt);
#endif
}

inline void cmp(my_struct *src, my_struct *dst) {
    unsigned int cond = 1;
    cond &= dst->uint1 == src->uint1;
    cond &= dst->uint2 == src->uint2;
    cond &= dst->uint3 == src->uint3;
    sassert(cond);
}

inline void cmp2(my_struct2 *src, my_struct2 *dst) {
    unsigned int cond = 1;
    cond &= dst->uint1 == src->uint1;
    cmp(&dst->st, &src->st);
#ifdef PTR
    cmp(dst->pt, src->pt);
#endif
    sassert(cond);
}
```

Listing A.3: Helper functions

In order to cover multiple common `PQUIC` use-cases at once, we defined two structures: `my_struct` which contains mainly unaligned bitfields and `my_struct2` which embed a `my_struct`. We also added optional structure pointer inside `my_struct2` in order to experiment such kind of initialization. Here are the tuning flags:

- **PTR**: Turn on inner structure pointer handling.
- **PTR_INIT**: Use a semi-explicit pointer initialization rather than relying on SeaHorn modeling. Here, the original structure lies on the stack of the main while its copy (see L.11 of `dup2` in Listing A.3) is stored on the heap.
- **FAIL**: Add a `sassert(0)` at the end of the main to ensure `unsat` correctness.

Those flags define the five CUA classes we tested: the S1X class experiments direct structure accesses, S2X class tests access to embedded structure fields and S3X tests embedded structure bit fields. S5X and S4X are respectively similar to S2X and S3X except they are accessing fields from a structure pointer. A sixth S0X class is added to ensure the whole code is behaving as expected e.g., triggers `unsat` with SeaHorn when no CUA is provided.

Results discussion Table A.1 shows the experiments results and they are clear. The solver produces the expected output e.g., `sat` result for each test case and the main issues concern the CEX execution. We observe two failures, either the CEX does not output the expected `__VERIFIER_error` message signaling that the error has been reached, either a segmentation fault is triggered during the CEX execution.

The first issue appears mainly when the CUA sets a field to 0 **without** pointer embedded in the structure[2]. Strangely, adding a pointer into `my_struct2` by enabling `PTR` flag seems to solve the issue besides one exception, the S35 test case which modifies a bitfield from a structure embedded into another with a value different from 0. It seems that the generated CEX associates 0 to each of the fields which fools its execution and does not trigger the error as expected. This contradiction between the CEX and the solver invalidates those `sat` outputs and makes them unusable for us.

That being said, this issue is easily solvable by explicitly initializing each variable by hand with something like shown in Listing A.4, it even makes S35 pass. Though, this fix is at the cost of additional code whose amount is proportional to the number of field per structure.

```
1 extern uint8_t sh_uint8();
2
3 my_struct2 s;
4 s.uint1 = sh_uint8_t();
5 s.st.uint1 = sh_uint8_t();
6 s.st.uint2 = sh_uint8_t();
7 s.st.uint3 = sh_uint8_t();
```

Listing A.4: Explicit structure initialization

The latter issue appears only when the inner pointer initialization is left to SeaHorn and applying the trick[3] advised by Gennari et al. [28] did not solve the issue. Replacing the embedded pointer initialization (L.4-5 of Listing A.2) by `s.c = sh_my_struct()` with the non-deterministic function defined as `extern my_struct* sh_my_struct()` even worsen the outcomes of Table A.1 i.e., all

[2] S10, S13, S20, S23, S30, S33

[3] Converting `foo->bar` pointer access to non-deterministic `get_bar(foo)` function call.

the cases with **PTR_INIT** flag enabled produce segfaults. Even if it does not come from the solver, this motivates the usage of at least semi-explicit pointer initialization.

Valgrind Running Valgrind on the CEX execution showed many `Conditional jump or move depends on uninitialised value(s)` e.g., for cases where no flags are set, such warning is triggered[4] for all the variables comparisons (L.18-20 and L.26 of Listing A.3). Test cases where only **PTR** flag is set clearly show, in addition to the previous warning, `Use of uninitialised value` and `Invalid read` warnings on the structure duplication code (L.2[5] of Listing A.3), the latter being the origin of the segfaults. Finally, Valgrind reports also `Conditional jump or move depends on uninitialised value(s)` warnings on some, but not all, variables comparisons when **PTR** and **PTR_INIT** flags are enabled.

Those observed behaviors are quite expected since none of the variables are explicitly initialized. The witness bitcode generated by SeaHorn will only attribute values to the variables leading to the `sat` state but not to the others ones. The segfaults are also explained by the accesses on a non-initialized pointer during structure duplication. Obviously, all the `uninitialised value(s)` warnings disappear when all variables are explicitly initialized in the `C` code with non-deterministic function calls.

Conclusion This example showed that all the tested fields configurations are well handled by SeaHorn but seems to provoke some corner cases in the CEX generation process. In order to circumvent this issue, we adopted a heavy notation which explicitly initialize all the structure's fields. This logic has to be repeated recursively on each structure or structure's pointer stored in a given structure.

[4]By using a `cond` variable, Valgrind only triggers warnings for the `sassert` call. In order to have more precise details, we replaced the `cond` manipulation by direct `sassert` calls e.g., `cond &= dst->a == src->a` became `sassert(dst->a == src->a)`.

[5]In reality, SeaHorn adds an additional `__seahorn_mem_load` instructions to model memory accesses [28]. It is this exact instruction which triggers the warnings but for readability we only point out the memory access performed in the `C` code.

CUA	Id	**PTR**	**PTR_INIT**	CEX
None	S00			0
	S01			1*
	S02	X		0
	S03	X	X	0
	S04	X		3*
	S05	X	X	1*
`s->uint1 = 0`	S10			2
	S11	X		3
	S12	X	X	1
`s->uint1 = 10`	S13			1
	S14	X		3
	S15	X	X	1
`s->st.uint1 = 0`	S20			2
	S21	X		3
	S22	X	X	1
`s->st.uint1 = 10`	S23			1
	S24	X		3
	S25	X	X	1
`s->st.uint2 = 0`	S30			2
	S31	X		3
	S32	X	X	1
`s->st.uint2 = 1`	S33			1
	S34	X		3
	S35	X	X	2
`s->pt->uint2 = 0`	S40	X		3
	S41	X	X	1
`s->pt->uint2 = 1`	S42	X		3
	S43	X	X	1
`s->pt->uint1 = 0`	S50	X		3
	S51	X	X	1
`s->pt->uint1 = 10`	S52	X		3
	S53	X	X	1

* `sat` forced with `sassert(0)` addition.
[0] No CEX required since expected outcome is `unsat`.
[1] CEX outputs `__VERIFIER_error was executed`.
[2] CEX **does not output** `__VERIFIER_error was executed`.
[3] Segmentation fault on CEX execution.

Table A.1: Structure initialization experiment results.

A.2 Double pointer

This example experiments with the manipulation of a dynamic sized array of structure pointers. It represents the kind of code we want to use in our `PQUIC` model in order to handle the `path` field of the `picoquic_cnx_t` structure.

Listing A.5 shows the source code of this example. Lines 35, 37, 45 and 46 are defined to try to remove some warning issues we experienced as this fix showed good results in single pointer check. We especially compare this code's behavior with either lines 37 or 38 enabled to understand their impact on the SeaHorn output.

Code at line 54 (memory free for real execution) has been removed to save some space in this document, please refer to the original `dbl_ptr.c` file to obtain full details. The structure and the helpers are defined in `simple_struct.h`.

Flags We defined multiple flags set at compile time in order to tune our example to show different behaviors. Here is their summary:

- **TEST** Enable SeaHorn specific code. Unset for real execution. Other flags are usable in conjunction with this one:
  - **ND_SIZE** Tell SeaHorn to initialize a non-deterministic size constrained to be at least 1. Otherwise, the size is hard-coded to 10.
  - **FAIL** Add a `sassert(0)` at the end of the main to ensure SeaHorn is still able to detect a simple failure.
- **CUA** Enable code chunk modifying a field of one of the structures. When enabled with SeaHorn, the verification should *always* `sat` and produce a valid CEX.

```
1  int main(void) {
2      unsigned int size;
3  #if defined(TEST) && defined(ND_SIZE)
4      size = sh_uint();
5      assume(size >= 1);
6  #else
7      size = 10;
8  #endif
9
10     my_struct **ptr, **ptr0;
11     ptr = (my_struct**) malloc(sizeof(my_struct*)*size); // init
12     for (int i=0; i<size; i++) {
13         ptr[i] = (my_struct*) malloc(sizeof(my_struct));
14         init(ptr[i]);
15     }
```

```
16
17     ptr0 = (my_struct**) malloc(sizeof(my_struct*)*size); // duplicate
18     for (int i=0; i<size; i++) {
19         ptr0[i] = (my_struct*) malloc(sizeof(my_struct));
20         dup(ptr[i], ptr0[i]);
21     }
22
23 #ifdef CUA // When defined should always sat
24     unsigned int o;
25 #if defined(TEST) && defined(ND_SIZE)
26     o = sh_uint();
27     assume(o < size);
28 #else
29     o = 1;
30 #endif
31     ptr[o]->a = 0; // Modifies one field of one of the structures
32 #endif // CUA
33
34     unsigned int cond;
35     my_struct **tmp = ptr, **tmp0 = ptr0; // comparison
36     for (int i=0; i<size; i++) {
37         cond = cmp(*tmp, *tmp0);
38         //cond = cmp(ptr[i], ptr0[i]);
39 #ifdef TEST
40         sassert(cond);
41 #else
42         printf("%i:%i\n%p:%p\n%p:%p\n\n", i, cond, ptr[i], *tmp, ptr0[i], *tmp0);
43 #endif
44         tmp++;
45         tmp0++;
46     }
47
48 #ifdef TEST
49 #ifdef FAIL
50     sassert(0); // Is SeaHorn still able to detect simple failure?
51 #endif // FAIL
52 #else
53 // > Snipped to save some space, see original file for more details.
54 #endif // TEST
55 }
```

Listing A.5: Double pointer experiment.

Results discussion Table A.2 shows the experiments we ran and their results. Unexpected outcomes are highlighted in red. Both S2 and S6 are the setups we are especially looking for since we want to prove a pluglet won't change any path information whatever is the number of path. We thus have to ensure that a

non-deterministic size is correctly handled by SeaHorn.

R0 and R1 indicate that our code is behaving as expected with both lines 37 and 38, which implies it is correct from a `C` standpoint. The memory accesses are asserted to be correct by `Valgrind` so they are not expected to skew the experimentation with SeaHorn.

We expect for S1 a `sat` result since we add explicit failing property into our code right after the verification loop. Interestingly, SeaHorn seems unable to reach this property for both comparison instructions, even with a hard-coded size. Hence, it makes the outcome of S0 doubtful since we are not able to prove that the `unsat` is legit. The most striking result we observe is S4 with line 38 enabled which shows a false negative result on a hard-coded size. For the rest of the experiments, the results are globally the ones expected for both lines 37 and 38 usage. We also consistently receive warnings about possible undefined values which are obviously wrong claims. This leaves some doubts on results accuracy since it indicates that SeaHorn maybe wrongly interpreted the code.

Execution	Id	**CUA**	**ND_SIZE**	**FAIL**	Expected	Result	
						L.37	L.38
Real	R0				$cond == 1\ \forall i$	valid	valid
	R1	X			$cond == 1\ \forall i \vert i\ != 1$	valid	valid
SeaHorn	S0				unsat	unsat	unsat
	S1			X	sat	unsat	unsat
	S2		X		unsat	unsat[1]	unsat[1]
TEST	S3		X	X	sat	sat[*]	sat[*]
set	S4	X			sat	sat[*]	unsat
	S5	X		X	sat	sat[*]	sat[*]
	S6	X	X		sat	sat[*,2]	sat[*,2]
	S7	X	X	X	sat	sat[*,3]	sat[*,3]

[*] CEX execution outputs `__VERIFIER_error was executed` (expected)
[1] `found 2 possible reads of undefined values` at L.37/L.38
[2] `found 1 possible reads of undefined values` at L.37/L.38
[3] `found 1 possible reads of undefined values` (no line precised)

Table A.2: Double pointer experiment results.

Conclusion All the false negatives we observe appear when we use a randomly (but “small enough“) chosen, hard-coded value for the array size which isn't a big deal since we are interested in another use-case. The main difference noted between lines 37 and 38 is a false negative on context modification which is not acceptable within our `PQUIC` model. Changing the pointer notation didn't remove the warnings about undefined reads but since the CEX were all able to trigger the

failed property, we are quite confident about the correctness of the results. That being said, we obtain such outcomes on a very simplistic structure containing only direct fields. Extending it with more complex field types could give different results but this is beyond the scope of this simple example.

A.3 Legitimate `unsat` on `sassert(0)`

We noticed that SeaHorn produces sometimes false negatives even on code with explicit failing assertions. For example, this has been observed for the experiment S1 of Appendix A.2. That being said, in some cases the failing assertions may be legitimately unreachable. Listing A.6 illustrates such legitimate behavior caused by an infinite loop.

```
1  int test(void) {
2      int i;
3      for (i=0; i<20; i++) {}
4      return i;
5  }
6
7  int main(void) {
8      int i = test();
9      sassert(i == 20);
10     while(1) {}
11     sassert(0);
12     return 0;
13 }
```

Listing A.6: Example of legitimate `unsat` with `sassert(0)` call

This code outputs `unsat` since the first assertion (line 11) is true and the second (line 13) is unreachable because of the infinite loop (line 12). Putting all the assertions after the infinite loop i.e., inverting line 11 and line 12, enforces this observation. Indeed, SeaHorn prints the following warning: `Possibly all assertions have been discharged by the front-end` which clearly indicates that the assertions were removed during the first optimization pass.

A.4 `UTHash` hashmap

The `UTHash` hashmap is widely used in the `PQUIC` structures but SeaHorn seems to have difficulties to handle its implementation. Listing A.7 shows the code we tested. From line 1 to 5, we defined the building block of the hashmap. It contains two elements: an integer id and a name encoded in a string. Line 7 declares the

hashmap as specified in the documentation[6]. With line 9 to 16, we declare a helper function to add a new item in the hashmap. Its code is heavily inspired by the one in the documentation[7].

```
typedef struct {
    int id;
    char name[10];
    UT_hash_handle hh;
} my_struct;

my_struct *users = NULL; /* important! initialize to NULL */

my_struct *add_user(int user_id, char *name) {
    my_struct *s;
    s = malloc(sizeof(my_struct));
    s->id = user_id;
    strcpy(s->name, name);
    HASH_ADD_STR(users, name, s); /* id: name of key field */
    return s;
}

extern unsigned int dummy___unsigned_int();
extern char dummy_char();

int main(void) {
    char name[5] = "test";

    sassert(users == NULL);
    sassert(HASH_COUNT(users) == 0);
    my_struct *in = add_user(10, name);
    sassert(users != NULL);
    sassert(HASH_COUNT(users) == 1);

    my_struct *out;
    HASH_FIND_STR(users, "test", out);
    sassert(out != NULL);
    sassert(in->id == out->id);

#ifdef FAIL
    sassert(0);
#endif

    return 0;
}
```

Listing A.7: test

[6]See https://troydhanson.github.io/uthash/userguide.html#_declare_the_hash.
[7]See https://troydhanson.github.io/uthash/userguide.html#_add_item.

In the main function, until line 27 all the assertions `unsat` as expected and declaring the **FAIL** flag at line 35 effectively cause `sat` outputs. However, each assertion upon line 27, tested individually, fail. Line 28 tries to ensure that a new element has actually been added in the hashmap[8] but SeaHorn is not able to prove that. Line 31 tries to get back a pointer toward the node whose key is "test"[9] and SeaHorn is neither able to prove that the pointer is non NULL nor that the ids are the same.

Since a such basic example is not well handled by SeaHorn, more complex usages of the hashmap structure, as the ones performed in `PQUIC`, are not expected to be functional.

[8]See `https://troydhanson.github.io/uthash/userguide.html#_count_items`.

[9]See `https://troydhanson.github.io/uthash/userguide.html#_string_keys`.

Appendix B

PQUIC structures

This appendix contains, for each internal `PQUIC` structure susceptible to be verified, the fields which **are considered or not** in our model. The fields are chosen according to our working hypothesis defined in Section 4.3.5. Those that are not mentioned here, because the justification of their verification is trivial[1], are all verified.

The notation used in each tabular is the following:

- **J** stands for Justification (**H** stands for Hypothesis, **C** stands for Corollary and **G** stands for Generation[2])
- **G** stands for Getter
- **S** stands for Setter (**D** stands for Direct and **I** stands for Indirect)
- **V** stands for Verified

We make a distinction between direct and indirect setters. The latter are defined as function first requiring a get call before being able to set a value. For example, in the `set_pkt_ctx` helper handling `picoquic_packet_context_t` structures, the modification of the `first_sack_item` field is not yet implemented (See Listing B.1).

```
[...]
case AK_PKTCTX_FIRST_SACK_ITEM:
    printf("ERROR: setting the first sack item is not implemented!\n");
    break;
[...]
```

Listing B.1: Sample of the `set_pkt_ctx` helper.

[1] For example, fields having a direct `set` function allowing their rewrite.

[2] It corresponds to fields additionally verified to ease the generation of the `PQUIC` model.

However, one could first call `get_pkt_cnx` to get the pointer toward the `first_sack_item` field. Then, using the `set_sack_item` helper, one could modify SACK item's fields. Obviously, the pointer toward the SACK item structure is not mutable but it is the case for the structure content. Verification is thus required.

B.1 `picoquic_packet_context_t`

Field type	**Field name**	**J**	**G**	**S**	**V**
`picoquic_sack_item_t`	*first_sack_item*	C2.1	X	I	X
`picoquic_packet_t*`	*retransmit_newest*	C2.1	X	I	X
	retransmit_oldest		X	I	X
	retransmitted_newest		X	I	X
	retransmitted_oldest		X	I	X
`plugin_struct_metadata_t*`	*metadata*	H5	X	D	

B.2 `picoquic_sack_item_t`

Field type	**Field name**	**J**	**G**	**S**	**V**
`struct st_picoquic_sack_item_t*`	*next_sack*	C2.1	X	I	X

B.3 `picoquic_packet_t`

Field type	**Field name**	**J**	**G**	**S**	**V**
`picoquic_packet_t*`	*previous_packet*	C2.1	X	I	X
	next_packet		X	I	X
`uint64_t`	*delivered_prior*	G	X		X
	delivered_time_prior		X		X
	delivered_sent_prior		X		X
`uint32_t`	*send_length*	G			X
`unsigned int`	*has_plugin_frames*	G			X
	delivered_app_limited		X		X
`picoquic_packet_plugin_frame_t*`	*plugin_frames*	H2			
`plugin_struct_metadata_t*`	*metadata*	H5	X	D	
`uint8_t []`	*bytes*	C2.1	X	D	

B.4 picoquic_cnx_t

Field type	Field name	J	G	S	V
`picoquic_quic_t*`	*quic*	H2			
`struct st_picoquic_cnx_t*`	*next_in_table* *previous_in_table*	H2			
`struct st_picoquic_cnx_id_t*`	*first_cnx_id*	H2			
`struct st_picoquic_net_id_t*`	*first_net_id*	H2			
`char const*`	*sni* *alpn*	 H2			
`picoquic_stream_data_cb_fn`	*callback_fn*	H2			
`void*`	*callback_ctx*	H2			
`uint8_t*`	*retry_token*	H2			
`struct st_picoquic_cnx_t*`	*next_by_wake_time* *previous_by_wake_time*	H2			
`void*`	*tls_ctx*	H2			
`struct st_ptls_buffer_t*`	*tls_sendbuf*	H2			
`picoquic_crypto_context_t[]`	*crypto_context*	C2.1	X	I	
`picoquic_congestion_algorithm_t const*`	*congestion_alg*	H2	X		
`queue_t*`	*reserved_frames* *retry_frames*	H2	X		
`queue_t* []`	*rtx_frames*	H2	X		
`protoop_plugin_t*`	*first_drr*	H2			
`plugin_request_t`	*pids_to_request*	H2	X		
`picoquic_stream_head*`	*first_plugin_stream*	H2			
`protocol_operation_struct_t*`	*ops*	H2			
`protoop_plugin_t*`	*plugins*	H2			
`plugin_struct_metadata_t*`	*metadata*	H5	X	D	
`protocol_operation_struct_t*`	*current_protoop*	H2			
`pluglet_type_enum`	*current_anchor*	H2			
`protoop_plugin_t*`	*current_plugin*	H2			
`protoop_plugin_t*`	*previous_plugin_in_replace*	H2			

B.5 picoquic_tp_t

Field type	Field name	J	G	S	V
`char*`	*supported_plugins* *plugins_to_inject*	H2 H2			

B.6 `picoquic_path_t`

Field type	Field name	J	G	S	V
`struct sockaddr_storage`	*peer_addr*	C2.1	X	D	
	local_addr		X	D	
`uint8_t []`	*challenge_response*	C2.1	X	D	
`void*`	*congestion_alg_state*	C2.1	X	D	
`picoquic_packet_context_t []`	*reset_secret*	C2.1	X	I	X
`plugin_struct_metadata_t*`	*metadata*	H5	X	D	

B.7 `picoquic_connection_id_t`

Field type	Field name	J	G	S	V
`uint8_t []`	*id*	C2.1	X		

Appendix C

PQUIC memory helpers misuse

The memory helpers provided by the PQUIC API allow pluglets to handle memory areas external to the PRE, i.e. in the host memory. The reachable areas should be limited to the ones really useful for plugins.

However, we found out that they allow full access, neither expected nor secure, to the plugin context. The method is the following:

1. A pointer is created in the host memory by calling `my_malloc`.
2. The address of the structure of interest is copied into this pointer with `my_memcpy`.
3. A direct access field is immediately readable and is writable by using `my_memset`.
4. An indirect field, i.e. another pointer contained in the structure of interest, is reachable by resuming to step 1.

We illustrate this misuse of the memory helpers with five examples. We use the `pquic/pquic` Docker container (hash `58cd2c2518cd`) for this. Listing C.1 and C.2 show the commands used to setup the PQUIC server and to launch the client which injects the plugins. They require that the current directory is the root of the repository containing the different example plugins[1].

```
1 docker run --rm --name quic_server -it -v $(pwd):/mnt pquic/pquic ./picoquicdemo -p
    5000
```

Listing C.1: Start a PQUIC server in the demonstration container.

```
1 docker exec -it quic_server bash -c './picoquicdemo -P /mnt/dummy_test/<plugin>.
    plugin localhost 5000 0:index.html' | grep Entering
```

Listing C.2: Start a client connection which inject the specified plugin.

[1] Their whole code is available at `https://github.com/nrybowski/pquic-formal-model/tree/master/dummy_test`.

C.1 cnx_ro_replace

Listing C.3 shows the single pluglet composing the first plugin. It is attached to the *replace* hook of the `is_ack_needed` protoop.

```
1 protoop_arg_t is_ack_needed(picoquic_cnx_t *cnx) {
2     PROTOOP_PRINTF(cnx, "Entering␣pluglet\t");
3     char **tmp = (char **) my_malloc(cnx, sizeof(void*));
4     my_memcpy(tmp, &cnx->sni, sizeof(void*));
5     PROTOOP_PRINTF(cnx, "SNI␣:␣%s\t", (protoop_arg_t) *tmp);
6     PROTOOP_PRINTF(cnx, "Exiting␣pluglet\n");
7     return (protoop_arg_t) 1;
8 }
```

Listing C.3: `cnx_ro_replace` pluglet.

The pluglet reads the `SNI` field of the `picoquic_cnx_t` passed as argument to the pluglet. This field is not expected to be reachable from the pluglet since there is no getter for it. Line 3 allocates a new pointer in the host memory. Line 4 copies the field of interest, here the address of the `SNI` string. Line 5 reads the field content.

Once injected, this pluglet outputs `Entering pluglet SNI : localhost Exiting pluglet`. This shows that a read access is possible on a field considered as unreachable in the plugin context.

C.2 cnx_rw_replace

This plugin extends the previous one. That is, it reads **and writes** the `SNI` field of the `picoquic_cnx_t` passed as argument to the pluglet. This field is not expected to be reachable from the pluglet since there is no getter **nor setter** for it.

```
1 protoop_arg_t is_ack_needed(picoquic_cnx_t *cnx) {
2     PROTOOP_PRINTF(cnx, "Entering␣pluglet\t");
3     char **tmp = (char **) my_malloc(cnx, sizeof(void*));
4     my_memcpy(tmp, &cnx->sni, sizeof(void*));
5     my_memset(*tmp+1, 'b', sizeof(char));
6     PROTOOP_PRINTF(cnx, "SNI␣:␣%s\t", (protoop_arg_t) *tmp);
7     PROTOOP_PRINTF(cnx, "Exiting␣pluglet\n");
8     return (protoop_arg_t) 1;
9 }
```

Listing C.4: `cnx_rw_replace` pluglet.

Listing C.4 shows the code of this pluglet. The main difference with the previous one is at line 5, where we replace the second byte of the `SNI` by “b“.

Upon execution, the pluglet outputs `Entering pluglet SNI : lbcalhost Exiting pluglet`. It clearly illustrates the write operation.

C.3 cnx_rw_pre

In order to ensure that the write performed in `cnx_rw_replace` is effective, a write is performed at the *pre* hook of `is_ack_needed` and a read is performed at the *replace* hook. The pluglet at *pre* hook is exactly the one shown in Listing C.4 except that “pluglet“ is replaced by “pre“ in the `printf`. At the *replace* hook, the pluglet from Listing C.3 is injected.

Here is the output once the plugin is fully executed `Entering pre SNI : lbcalhost Exiting pre Entering pluglet SNI : lbcalhost Exiting pluglet`. This confirms that the write operation in the plugin context is an actual write which persists among pluglets executions.

C.4 plugin_rw_replace

With the three previous plugins, we showed that read and write accesses are possible on fields from the plugin context. Now, we try to access the plugin data stored in the `current_plugin` field of the `picoquic_cnx_t` structure. This field is a pointer to a `protoop_plugin_t` structure which contains actual plugin's data such as its name, the path to its manifest on the host, the hash of its name used in a hashmap, etc.

```
1  protoop_arg_t update_ack_delay (picoquic_cnx_t *cnx) {
2      PROTOOP_PRINTF(cnx, "Entering pluglet\t");
3      protoop_plugin_t **tmp = (protoop_plugin_t**) my_malloc(cnx, sizeof(void*));
4      tmp = my_memcpy(tmp, &cnx->current_plugin, sizeof(void *));
5      PROTOOP_PRINTF(cnx, "Plugin name:<%s>\t", (protoop_arg_t) (*tmp)->name);
6      char **tmp2 = (char**) my_malloc(cnx, sizeof(void*));
7      tmp2 = my_memcpy(tmp2, &((*tmp)->path), sizeof(void*));
8      PROTOOP_PRINTF(cnx, "Path:<%s>\t", (protoop_arg_t) *tmp2);
9      PROTOOP_PRINTF(cnx, "Before return\n");
10     return 0;
11 }
```

Listing C.5: `plugin_rw_replace` pluglet.

Listing C.5 shows the pluglet code implementing the `update_ack_delay` protoop. The pluglet is attached to the *replace* hook of the protoop. Lines 3 and 4 allocate a pointer in the host memory and copy the address of the `current_plugin` pointer. Line 5 prints the name of the plugin. Lines 6 and 7 allocate a new pointer and copy the path to the manifest of the plugin. Finally, line 8 prints the path content.

Once executed, this plugin outputs:
`Entering pluglet Plugin name:<be.nrybowski.plugin_rw_replace> Path:</mnt/dummy_test/plugin_rw_replace.plugin> Before return`.

Those values are effectively the real plugin name and the real path toward its manifest in the container.

C.5 cnx_rw_tested_replace

This last plugin (shown in Listing C.6) rewrites a context input for the `update_ack_delay` protoop at its *replace* hook. In this case, there is no need to copy the pointer address as we reset it directly to 0 at line 4.

```
protoop_arg_t update_ack_delay(picoquic_cnx_t *cnx) {
    PROTOOP_PRINTF(cnx, "Entering pluglet\t");
    PROTOOP_PRINTF(cnx, "Before %u\t", (uint64_t) get_cnx(cnx, AK_CNX_INPUT, 0));
    my_memset(&cnx->protoop_inputv[0], 0, sizeof(void*));
    PROTOOP_PRINTF(cnx, "After %u\t", (uint64_t) get_cnx(cnx, AK_CNX_INPUT, 0));
    PROTOOP_PRINTF(cnx, "Exiting pluglet\n");
    return (protoop_arg_t) 1;
}
```

Listing C.6: `cnx_rw_tested_replace` pluglet.

The pluglet outputs `Entering pluglet Before 924385483 After 0 Exiting pluglet` upon execution. It clearly shows that the first context input is re-written.

Bibliography

[1] BTF kernel documentation. https://www.kernel.org/doc/html/v5.12/bpf/btf.html. [Online. Accessed: 2021-05-03].

[2] Discussion about eBPF verifier extension with new socket type and btf usage. https://lore.kernel.org/bpf/20200922040830.3iis6xiavhvpfq3v@ast-mbp.dhcp.thefacebook.com/. [Online. Accessed: 2021-05-03].

[3] eBPF safety overview. https://ebpf.io/what-is-ebpf/#ebpf-safety. [Online. Accessed: 2021-06-08].

[4] Linux kernel documentation. eBPF verifier limitations. https://www.kernel.org/doc/html/v5.12/bpf/bpf_design_QA.html#q-what-are-the-verifier-limits. [Online. Accessed: 2021-06-08].

[5] Modified uBPF VM for PQUIC. https://github.com/p-quic/ubpf. [Online. Accessed: 2021-03-17].

[6] Seahorn: Cex generation tuning. https://github.com/seahorn/seahorn/tree/master/sea-rt. [Online. Accessed: 2021-04-10].

[7] SeaHorn: Inconclusive result on solver and CEX contradiction. https://github.com/seahorn/seahorn/issues/61#issuecomment-268038497. [Online. Accessed: 2021-04-10].

[8] Seahorn verification framework. https://seahorn.github.io/. [Online. Accessed: 2021-03-04].

[9] SeaHorn's BMC engine tuning. https://gitter.im/seahorn/seahorn?at=5c2635a6db5b5c68831d5035. [Online. Accessed: 2021-04-10].

[10] Seccomp project. Linux kernel documentation. https://www.kernel.org/doc/html/v5.12/userspace-api/seccomp_filter.html. [Online. Accessed: 2021-06-08].

[11] uBPF VM's source code. https://github.com/iovisor/ubpf. [Online. Accessed: 2021-03-04].

[12] Specification of Internet Transmission Control Program. RFC 675, December 1974.

[13] in-toto Linux Foundation project. https://in-toto.io/, 2020. [Online. Accessed: 2021-06-08].

[14] Sigstore Linux Foundation project. https://sigstore.dev/what_is_sigstore/, 2021. [Online. Accessed: 2021-06-08].

[15] The Update Framework (TUF). https://theupdateframework.io/, 2021. [Online. Accessed: 2021-06-08].

[16] Ken Borgendale Andrew Banks, Ed Briggs and Rahul Gupta. MQTT Version 5.0. https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html, 2019. [Online. Accessed: 2021-06-01].

[17] Gurfinkel Arie. SeaHorn's CHC engine tuning. https://gitter.im/seahorn/seahorn?at=5c26391f37975e7ca9391fab, 2018. [Online. Accessed: 2021-04-10].

[18] Zuzana Baranová, Jiří Barnat, Katarína Kejstová, Tadeáš Kučera, Henrich Lauko, Jan Mrázek, Petr Ročkai, and Vladimír Štill. Model checking of c and c++ with divine 4. In *International Symposium on Automated Technology for Verification and Analysis*, pages 201–207. Springer, 2017.

[19] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, page 46. California, USA, 2005.

[20] Armin Biere. Bounded model checking. In *handbook of Satisfiability*, pages 739–764. IOS Press, 2021.

[21] Sharon Boeyen, Stefan Santesson, Tim Polk, Russ Housley, Stephen Farrell, and Dave Cooper. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, May 2008.

[22] Marc Brockschmidt and Heidy Khlaaf. T2 Temporal Prover. http://mmjb.github.io/T2/, 2019. [Online. Accessed: 2021-06-01].

[23] Lucas Cordeiro. Detection of software vulnerabilites: Static Analysis. https://ssvlab.github.io/lucasccordeiro/courses/2020/01/software-security/slides/lecture03.pdf, 2020.

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

[25] Quentin De Coninck, François Michel, Maxime Piraux, Florentin Rochet, Thomas Given-Wilson, Axel Legay, Olivier Pereira, and Olivier Bonaventure. Pluginizing quic. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 59–74. 2019.

[26] Developers. Terrform provider for Libvirt. `https://github.com/dmacvicar/terraform-provider-libvirt`, 2021. [Online. Accessed: 2021-06-01].

[27] Eclipse Foundation. Paho MQTT. `https://www.eclipse.org/paho/`, 2019. [Online. Accessed: 2021-06-01].

[28] Jeffrey Gennari, Arie Gurfinkel, Temesghen Kahsai, Jorge A Navas, and Edward J Schwartz. Executable counterexamples in software model checking. In *Working Conference on Verified Software: Theories, Tools, and Experiments*, pages 17–37. Springer, 2018.

[29] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and precise static analysis of untrusted linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1069–1084, 2019.

[30] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. The seahorn verification framework. In *International Conference on Computer Aided Verification*, pages 343–361. Springer, 2015.

[31] Arie Gurfinkel and Jorge A Navas. A context-sensitive memory model for verification of c/c++ programs. In *International Static Analysis Symposium*, pages 148–168. Springer, 2017.

[32] HashiCorp. Packer. `https://www.packer.io/`, 2021. [Online. Accessed: 2021-06-01].

[33] HashiCorp. Terraform. `https://www.terraform.io/`, 2021. [Online. Accessed: 2021-06-01].

[34] Red Hat. Libvirt: Virtualization API. `https://libvirt.org/`. [Online. Accessed: 2021-06-01].

[35] Jana Iyengar and Martin Thomson. Quic: A udp-based multiplexed and secure transport. *Internet Engineering Task Force, Internet-Draft draftietf-quic-transport-17*, 2018.

[36] Maya Kaczorowski. Secure at every step: What is software supply chain security and why does it matter? `https://github.blog/2020-09-02-secure-your-software-supply-chain-and-protect-against-supply-chain-`, 2020. [Online. Accessed: 2021-06-08].

[37] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230. Dttawa, Dntorio, Canada, 2007.

[38] Ben Laurie, Adam Langley, and Emilia Kasper. Certificate Transparency. RFC 6962, June 2013.

[39] Roger A Light. Mosquitto: server and client implementation of the mqtt protocol. *Journal of Open Source Software*, 2(13):265, 2017.

[40] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX winter*, volume 46, 1993.

[41] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[42] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.

[43] John Moy. OSPF Version 2. RFC 2328, April 1998.

[44] Python Core Team. Python 3.9.2. `https://www.python.org/`, 2021.

[45] Yakov Rekhter, Tony Li, Susan Hares, et al. A border gateway protocol 4 (bgp-4), 1994.

[46] Eric Rescorla and Tim Dierks. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, August 2008.

[47] Marta Rybczyńska. Bounded loops in BPF for the 5.3 kernel. `https://lwn.net/Articles/794934/`, 2019. [Online. Accessed: 2021-06-01].

[48] Chamarty Sitaram. Gitolite. `https://gitolite.com/gitolite/`, 2021.

[49] Thomas Wirtgen, Quentin De Coninck, Randy Bush, Laurent Vanbever, and Olivier Bonaventure. xbgp: When you can't wait for the ietf and vendors. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, pages 1–7, 2020.

[50] Ilja S Zakharov, Mikhail U Mandrykin, Vadim S Mutilin, EM Novikov, Alexander K Petrenko, and Alexey V Khoroshilov. Configurable toolset for static verification of operating systems kernel modules. *Programming and Computer Software*, 41(1):49–64, 2015.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**