Enabling Application-Specific Programmable Compute Infrastructure

by

Greg Cusack

B.S., Computer Science and Engineering, Santa Clara University, 2016

B.S., Electrical Engineering, Santa Clara University, 2016

M.S., Electrical Engineering, University of Colorado Boulder, 2020

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Electrical, Computer, and Energy Engineering

2022

Committee Members:

Eric Keller, Chair

Eric Rozner

Eric Wustrow

Tamara Silbergleit Lehman

Joe Izraelevitz

Cusack, Greg (Ph.D., Computer Engineering)

Enabling Application-Specific Programmable Compute Infrastructure

Thesis directed by Professor Eric Keller

As the demand for cloud computing services grows, developers are looking for ways to increase the security, squeeze out the highest performance, and achieve the lowest costs for their applications. Applications require a thought out security process – e.g. how to use secure hardware to isolate sensitive computations from an untrusted operating system. Network monitoring systems are needed to monitor network traffic flowing in and out of an application. Applications also require careful CPU and memory allocations for the business logic running in containers at the application layer.

In order to manage these three components of an application, cloud providers provide an application control and management layer that serves as an entrypoint into any application. Developers can utilize the secure hardware features to enable secure, verifiable computing. Developers can also monitor the network traffic traversing their application's network and manage CPU and memory allocations. Unfortunately, despite all of these tools to deploy, monitor, and secure an application, developers still lack the controls required to optimize their specific applications' security, performance, and efficiency.

In this thesis, we first explore and outline the root cause of this rigidity and lack of control prevalent in today's cloud. We focus on rigidity in the three areas of secure hardware, network monitoring, and compute resource allocation. We then build new, programmable platforms and systems that allow users to design and implement application-specific secure hardware features, network monitoring applications, and compute resource allocation algorithms. We evaluate each of our systems and show how a developer can now optimize their applications with fine-grained control over the underlying compute infrastructure.

Dedication

To my family: Mom, Dad, Christopher, and Brennan. I love you. Thank you for all of the time you put in to make me succeed. I owe everything to you.

Acknowledgements

Thank you to my mom for instilling drive, passion, discipline, and a sense of humor into me, and for all the hours quizzing me and helping me study for my middle/high school tests. To my dad for teaching me accountability, integrity, logical thinking, and having the patience to teach me how to write. To Christopher, for giving me a role model to look up and aspire to. To Brennan, for supporting me through literally everything, being my biggest cheerleader, and teaching me to be comfortable in my own skin. To Lila, you've always stood by me in the toughest days throughout my research. You never stopped supporting, encouraging, and pushing me. To Shoba Krishnan, for encouraging me to get my Ph.D., believing in me, inspiring me, and connecting me to so many good and brilliant people. To Darren Atkinson, for instilling a love of computer science into me, advising me, and spending his Thanksgiving writing a letter of recommendation for me for CU Boulder. To my friends I consider family from back home and in undergrad, you kept me laughing and grounded in the toughest times. To Oliver Michel, you taught me everything I know about C++ and building high quality systems and code. To Sepideh Goodarzy and Maziyar Nazari, for your friendship and your brilliant work. To Aimee Coughlin, Azzam Alsudais, Prerit Oberai, Marcelo Abranches, Mohammad Hashemi, Erika Hunhoff, Zaid Alali, Karl Olson, Dwight Browne, Max Hollingsworth, and Eric Wustrow thank you for your work, your ideas, and your friendship. To Eric Keller, my advisor, mentor, professor, friend, etc, I could fill up infinity pages of all the things you've done for me. You always supported, believed in, and pushed me. You kept me excited about research and always eliminated my creeping thoughts of imposter syndrome. I would never have completed this thesis without you. I am forever thankful for you and your support.

Contents

Chapter

Tables

Table

Figures

Figure

Chapter 1

Introduction

Due to the growing demand for cloud computing services and projected rise in dollars spent on cloud computing services [48], cloud and datacenter companies are constantly looking for ways to support the influx of more users. Cloud computing platforms are designed for virtually everyone, meaning providers are striving to make deploying and running applications as simple and general as possible. At the same time, developers are constantly looking for ways to increase the security, squeeze out the highest performance, and achieve the lowest costs for their applications. To this end, cloud providers expose tools that allow users to control and manage the underlying hardware and software systems. Control over the hardware gives developers the ability to optimize the security, performance, and efficiency of their applications.

Cloud infrastructure is composed of multiple layers of abstraction all layered on top of each other. Figure 1.1 shows a simplified, high-level view of a typical cloud architecture setup. At the bottom of Figure 1.1 lies the cloud's compute hardware distributed over a set of physical servers. This layer includes CPUs, NICs, memory modules, GPUs, secure hardware, etc housed within racks connected via a datacenter-wide network. The next layer up consists of the compute resource pool abstraction, typically comprised of a hypervisor with an overlaying operating system. The next layer up is the application layer, where developers can deploy their application-level code.

In the cloud, an application consists of more than just the business logic running at the application layer. Figure 1.1 also shows a few key areas that developers must address when deploying an application in the cloud. Applications require careful CPU and memory allocations for the

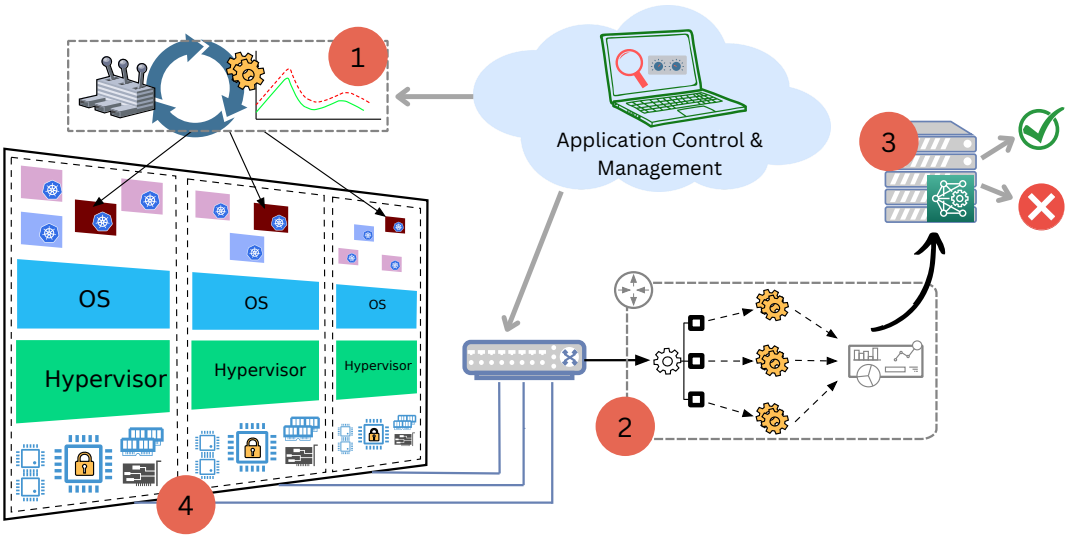


Figure 1.1: Cloud Architecture and Abstractions

business logic running in containers at the application layer ①. Furthermore, network monitoring systems must be setup to monitor network traffic flowing in and out of an application ②. Developers can then build network intrusion detection systems on top of the network monitoring platform in order to secure their network and application ③. Applications also require a thought out security process. Developers can use secure hardware to isolate sensitive computations from an untrusted operating system ④. In this thesis, we focus on these four components of any cloud application, secure hardware, network monitoring, and compute resource allocation.

At the top of Figure 1.1, we can see the application control and management later that serves as an entrypoint into any application. This layer is provided to developers by the cloud providers and is composed of a collection of tools that are general and designed to be simple to use. Through this layer, developers can do a few things. First, developers can deploy containers and allocate compute resources to their application. They can also build network monitoring application to monitor traffic traversing their application's network. Developers can also utilize the secure hardware features to enable secure, verifiable computing on an untrusted platform. It is important to note that the application control anad management layer is defined by the cloud provider,

meaning they dictate to what extent you can access the underlying systems. Unfortunately, despite all of these tools to deploy, monitor, and secure an application, developers still lack the controls required to optimize their specific applications' security, performance, and efficiency.

1.0.1 Rigidity in the Cloud Today

We can see this lack of control and flexibility in the largest cloud provider, AWS. AWS Nitro Enclaves, is AWS' secure hardware offering [24]. This allows users to run applications that need secure hardware in the cloud. Nitro Enclaves provide isolated computing environments that can protect and securely process highly sensitive data (e.g. health data, proprietary information, etc). Nitro Enclaves also support verifiable secure enclave computing, meaning that AWS will validate that the code that you want them to run actually runs within the enclave [335]. Despite secure hardware support in AWS, Nitro Enclaves still fail to enable security optimization for cloud applications. For example, Nitro Enclaves do not support remote attestation with a flexible root of trust [24]. This means that Amazon is the root to attest that Amazon is running what you want Amazon to run. In other words, when you give your enclave code to Amazon, you have to trust Amazon is running your enclave code because Amazon just verifies with themselves that they ran your code properly. On top of the need to trust AWS, since Amazon controls the secure hardware, you are at the whim of Amazon to provide enclave updates, new features, and bug fixes. This lack of control, from a developer standpoint, over a key security component of an application results in poorly optimized application security.

AWS also supports network monitoring for developers' applications with AWS CloudWatch [57]. However, AWS CloudWatch only exposes eight different network statistics to developers – in/out byte and packet counts and in/out byte and packet drops [35]. With CloudWatch, users can only create simple alarms based on packet and byte counts. User's have no ability to perform a root-cause analysis of their network nor build security applications based on network traffic. Building any network intrusion detection system would result in poor performance as the granularity of information one can monitor is vastly limited. The lack of insight into the network prevents users

from optimizing their applications' network security, performance, and efficiency.

Next, AWS EKS, one of AWS' container platforms, supports vertical autoscaling, where containers scale as compute demands change [11]. Autoscaling eases the burden of setting container limits precisely, improving both performance and efficiency. Unfortunately, despite autoscaling support, AWS EKS still typically requires container resources to be manually adjusted. EKS is also slow to react to workload changes and requires a container restart in order to scale [21]. EKS' heavyweight and manual scaling results in poorly optimized application performance and efficiency.

So, the question remains, why are the tools and abstractions provided to developers to control the underlying software and hardware systems so rigid and inflexible?

1.0.2 Rigidity comes from the underlying software and hardware systems

The problem of rigidity does not necessarily lie with the cloud providers themselves. In fact, the underlying platforms that cloud providers build on top of simply do not support a high level of flexibility. As a result, cloud providers cannot provide the flexibility to the end user if the underlying hardware or software systems do not support it. The rigidity of the underlying hardware and software systems results in a lack of application control at the cloud provider level, leading to poorly optimized application security, performance, and efficiency.

1.0.3 Flexible underlying hardware and software systems

We asked ourselves, what if we could create and then expose that flexible underlying platform to developers? For developers, a high level of flexibility is great because they can then finely control the underlying compute systems themselves. Fine-grained control would also enable them to optimize their specific applications' security, performance, and efficiency. At the same time, high flexibility benefits the cloud provider as well; they can then build abstractions on top of our implementations as they see fit for their business.

In this dissertation, we first identify the shortcomings and rigidity of the underlying hardware and software systems that control and manage secure hardware, network monitoring, and compute

resources. We then build new, programmable platforms and systems that allow users to design and implement application-specific secure hardware features, network monitoring applications, and compute resource allocation decisions.

The rest of this dissertation is organized as follows. Chapter 2 dives into the area of secure hardware, and our flexible, reconfigurable secure hardware. Chapter 3 details our work in the area of network monitoring and our solution of packet-level, network analytics at cloud scale. Chapter 4 investigates the efficacy of machine learning-based and neural network-based network intrusion detection systems build on top of packet-level network monitoring systems. Chapter 5 looks at a event-based, sub-second container resource allocation system that enables both high containerized application performance and efficiency in the cloud. In each of the chapters mentioned above, we first identify the current research and relevant work in the area. We then show how the rigidity of the underling platform prevents application security, performance, and/or efficiency optimization. Finally, we present our solution that enables users to define customized, application-specific systems to optimize their applications' security, performance, and/or efficiency. This dissertation then wraps up with a conclusion and a discussion around future directions and research.

Chapter 2

Enabling Programmable Secure Hardware

We begin the core of this dissertation by looking at the rigidity of secure hardware. The current state of fixed silicon, secure hardware prevents developers from optimizing their applications' security. To combat this rigidity, we provide a novel solution that allows developers to optimize their applications' security through reprogrammable secure hardware.

Modern CPU designs are beginning to incorporate secure hardware features, enabling new applications that take advantage of them. However, implementing these secure functions in hardware is expensive, time consuming, and makes it difficult to update when vulnerabilities are discovered or new features are desired. Because of this expense, only a few large companies have entered the market, leaving system developers that use secure hardware little choice in the set of features they can use. Furthermore, developers have also found themselves at the whim of the hardware providers. Developers must rely on the providers to push updates, add new features, and fix security issues, some of which cannot be fixed due to the immutable nature of silicon. The inability for developers to define secure hardware features for their specific applications results in poorly optimized application security.

We see an alternative to this ecosystem using **reprogrammable logic** (i.e. FPGAs) that are increasingly integrated into traditional computational systems such as data centers and embedded systems. In this chapter, we identify and overcome several challenges to using commodity reprogrammable logic for implementing secure hardware. We present a framework for leveraging FPGAs along with a minimal amount of fixed hardware already present in many systems that enables arbi-

trary secure functions to be designed. Because these systems are implemented in reprogrammable hardware, they can be custom designed, built, and manufactured with significantly lower overhead and expense, while achieving the same security as fully silicon-based secure hardware.

To demonstrate the flexibility of our alternative architecture, we implement several proof-of-concept secure hardware functions on our platform, including a secure co-processor enclave similar to Apple's Secure Enclave, and a remote attestation system similar to Intel's SGX. We show that these designs are practical, enabling secure applications with a modest performance overhead, but at significantly lower cost and higher flexibility compared to existing silicon-based implementations.[1]

2.1 Introduction

Secure hardware provides many benefits for securing computing systems. It enables encrypting sensitive data where physical access to the device is required to decrypt it [67], authenticating data feed systems [453], scaling blockchain transactions [300], and has the promise to address many of the security challenges with cloud computing [168]. However, despite the potential benefits, we are stuck with a constrained ecosystem of secure hardware providers.

Due to the cost, time, and complexity of designing and manufacturing proceessor hardware [64, 63], the design choices and trade-offs are decided unilaterally by the small set of chip manufacturers. This results in scattered support of a wide range of features, and ultimately limited selection for users of secure hardware. Table 2.1 presents a summary of several secure hardware systems and the features they choose to support. Even in this modest set of features, there is no existing system that offers every feature, despite each system implementing features the other does not.

Furthermore, updates to secure hardware systems in response to discovered vulnerabilities [179, 379, 438, 431, 430, 100, 134, 135] or demand for new features are at worst impossible, and at best gated by the chip manufacturers, leaving system designers that use secure hardware at

[1] Work published at FPGA 2019 [202]

Feature	TPM	TZ	SGX
Flexible Root of Trust	●	●	○
TEE	○	●	●
Remote Attestation	●	○	●
Peripheral Access	○	●	○
Trusted Input	○	◐	○
Hardware RNG	●	○	●
Hardware Crypto	●	◐	◐
Secure Storage	●	○	●
Shared Architecture	◐	●	●
Oblivious Memory	○	○	●
Cache SC Defense	●	○	○
TLB SC Defense	○	●	○

Table 2.1: Comparing the features supported by Trusted Platform Modules (TPMs), ARM TrustZone (TZ), and Intel SGX. ● represents support, ◐ represents partial support or support that depends on how the design is instantiated, and ○ represents no support.

the mercy of a few companies.

In this chapter, we seek to empower the individuals that ultimately use secure hardware to make decisions that are right for their needs, rather than the hardware manufacturers making choices for them.

Prior research has proposed that programmable hardware, such as field-programmable gate arrays (FPGAs), are suitable for implementing security functions [201, 240, 315, 406, 369, 318, 223, 196, 213]. FPGAs are programmable, providing flexibility to define the exact features that are needed, while allowing updates and retaining the performance benefits of hardware [201, 240]. Importantly, FPGAs are no longer special purpose devices, but becoming pervasive in computing platforms such as cloud computing (**e.g.,** Amazon [9] and Microsoft data centers [91, 190, 358]), and in embedded systems for which secure hardware can provide great benefits, such as self-driving cars [29].

The programmable nature of FPGAs, however, raises a significant concern with regards to using them as a basis for realizing secure hardware – an attacker can read or modify the contents of the FPGA. This is in contrast to secure hardware systems built into silicon, which are "fixed", and cannot have their functionality changed after manufacture. Modern FPGAs include hardware that supports encrypted bitstreams [79, 434]. While an improvement, we argue that this doesn't

completely solve the problem, but this only reduces the control of reprogrammability to a single party. This party is responsible for generating and maintaining the keys that protect access and functionality of the device. In other words, it depends on human / business processes, which, as history has shown with the frequent password and other data leaks [432] (including secure boot keys [100]), cannot be counted on.

In this chapter, we introduce a novel mechanism to address this problem where we build on the capabilities provided by modern FPGAs and **put the device itself in control over the programmability**, thus removing the trust dependence on a third party's processes and providing developers with control over how the secure hardware is protected. This consists of two key aspects. The first is a self-provisioning mechanism where a device is initially brought up in a provisioning configuration, and then internally generates keys, and reprograms itself using these keys. In this way, the keys which control the configuration of the FPGA are only accessible internal to the device. The second is a policy driven update mechanism, where the hardware running in the FPGA is programmed with a policy which determines under what conditions to allow an update. In this way, we empower the secure hardware developer with the choice for how updates can occur (which could include a policy to block all updates). This allows the developer to choose (and commit to) how updates are (or aren't) performed on the device, allowing them to decide between a locked-down design similar to silicon-based secure hardware, or leaving systems flexible once deployed.

We demonstrate that this new mechanism is practical today with off-the-shelf FPGAs. Our implementation uses the Xilinx Zynq UltraScale+ MPSoC FPGA on the ZCU102 board. The application of this is broad, but as a single running example, we implement a secure coprocessor with an Intel SGX-like remote attestation feature. Unlike SGX's attestation, our remote attestation is designed to allow the device provisioner to choose who the root of trust is (rather than Intel's fixed root of trust being Intel), allowing for a wider range of trusted third parties to enable verified remote execution. We further use this running example to enable updates, which are motivated in this case to enable a response to newly discovered vulnerabilities, such as Spectre [287]. We provide

an SDK to compile programs to execute in this secure co-processor environment. Unique to this FPGA environment, we can compile the developer's C code to either hardware using high-level synthesis, or to software to run on a soft processor (a CPU implemented using the FPGA logic). We built two applications on top of this customized secure co-processor – a password manager (similar to the example in the Intel SGX tutorial), and a contact matching application (emulating the SGX-enabled private contact discovery service operated by Signal [312]).

In the remainder of this chapter we first discuss the past efforts of secure hardware on FPGAs (Section 5.2). We then provide an overview of the system architecture, threat model, and motivating example in Section 5.3. We describe the the architecture in Sections 2.4 and 2.5. We then describe the implementation of the self-provisioning and secure update mechanism (Section 5.5) and the secure co-processor with remote attestation (Section 2.7). We wrap up with evaluation (Section 5.6), and conclusions and future work (Section 2.10).

2.2 Past Attempts (and why process trust matters)

In this chapter we propose using FPGAs as a platform to build secure hardware. Here, we discuss past works, and identify the key unmet challenge in reaching this goal.

2.2.1 Security Functions on an FPGA

The idea of implementing security functions on an FPGA is not new. In fact, it has been proposed for decades. Research has been published on everything from network security applications (**e.g.,** firewalls [315] and intrusion detection [406]) to cryptographic algorithms [213]. More recently, and highly related to our motivating examples, the SAFES architecture demonstrated the use of FPGA components to provide security primitives and guarantee invariants in program execution [240], and Sanctum is a RISC ISA extension realized on an FPGA that mitigates software side-channels and protects DRAM access [201].

Although these examples demonstrate the ability to implement security functions on an FPGA, they do not address the somewhat obvious threat of an adversary who reprograms the

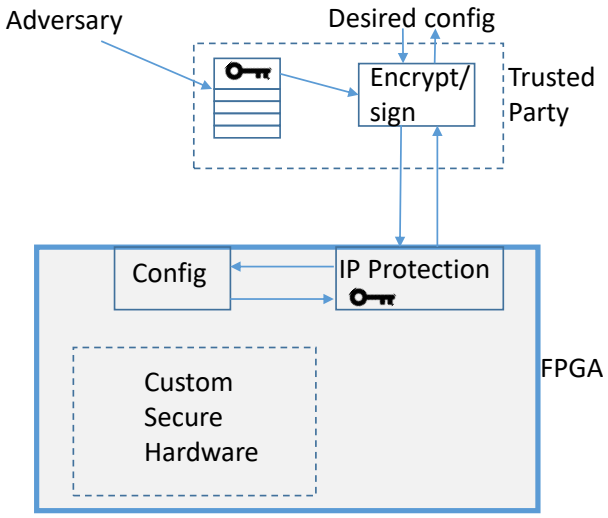


Figure 2.1: Custom secure hardware on an FPGA with IP protection: A designated party shares a cryptographic key with the FPGA which is used to ensure only FPGA configuration signed/encrypted with this key can re-program the FPGA. The designated party uses processes to protect the storage of the key, but an adversary can attack those processes and gain access to the shared key.

FPGA, changing the device configuration and functionality. We argue that for many secure hardware applications, this is a particularly important threat to address. For instance, if a device manufacturer wishes to offer remote attestation features (such as in Intel SGX) or hardware-protected keys for hardware security modules (HSMs), their design must protect against an adversary with physical (or remote) control over the device after its initial configuration.

By default FPGA's provide no protection to their configuration, allowing an adversary to read or reprogram whatever functionality is placed in it, allowing them to read out sensitive keys or change the device's behavior.

2.2.2 Security Functions with Bitstream Encryption

In response to this, FPGA manufacturers introduced bitstream protection technology, whether for intellectual property (IP) protection or specifically to support secure hardware [79, 434]. As illustrated in Figure 2.1, a third party programs a key into the FPGA and then maintains that key (external to the FPGA) so that it can be used to create an FPGA configuration that is encrypted and/or signed. In this way, knowledge of that key is needed to program the FPGA or read its

configuration.

While an improvement, it fundamentally depends on a human-driven / business process to protect the key that is programmed into the FPGA. Unfortunately, this has proven to be a challenging problem and particularly fragile means for security. We have seen countless data leaks, including passwords [432] and even secure boot keys [100] (things that we **should** be able to assume won't be leaked). In addition, governments can compel key-holders to divulge their secrets in order to attack individuals, such as in the FBI vs. Apple [44], ultimately undermining end-user trust in the systems. In short, IP protections only serve to focus an adversary's efforts on the process, and once successful would still be able to read or modify any FPGA that was under the 'protection' of that party.

2.3 System Architecture

2.3.1 High-level Overview

We present our high-level design which eliminates the human / business processes from the trust chain. We do this by designing the FPGA to have control over its own reprogrammability, and allowing it to determine when (or if) to allow updates to itself. This design eliminates the need for a trusted party to maintain keys through a business processes, which we argue has historically been shown to be problematic.

The self-provisioning system is designed to allow the device to be initially provisioned once by a system manufacturer into a secure state, and thereafter prevent any future updates externally. To do this, we leverage existing secure hardware systems used for IP protection (**e.g.,** secure boot) that controls the boot process of the device. We configure the secure boot to only allow a single configuration to be loaded into the FPGA. This configuration effectively locks out external access, preventing an adversary with physical access from changing the hardware loaded into the FPGA. Once in this state, **not even the original manufacturer can directly change the configuration**. The private keys used to sign this configuration are generated on the FPGA

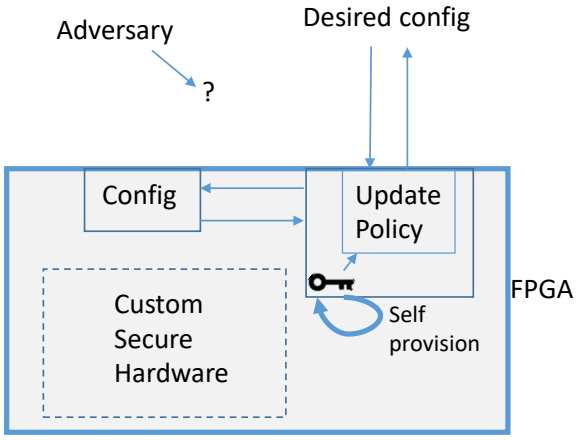


Figure 2.2: Secure Hardware on an FPGA with Self-Provisioning and Secure Updates. As the keys are only held within the FPGA, and updates are governed by hardware that implements an update policy, an adversary cannot gain access to the key or re-program the FPGA.

during provisioning, and stored in a secure storage that is only accessible to the FPGA itself once booted. Because secure boot prevents loading arbitrary bitstreams into the FPGA, nothing except the FPGA itself has access to the secret keys needed to sign new bitstreams.

This self-provisioning process prevents any future updates from being applied from an external source, but still allows the device itself to authorize and apply updates. We note that a developer could decide to disallow updates entirely by programming a configuration that simply discarded its own key, and gain the benefits of silicon-based secure hardware. However, should the developer wish to leverage the reprogrammability of the FPGA, they can choose to do so. If they do, the FPGA is configured with a subsystem for authorizing and applying updates to itself. This subsystem can implement security policies that are more powerful than simply giving up a remote key to the manufacturer. For example, in addition to a signed update from the manufacturer, the subsystem could determine if it is currently in a certain unlocked or safe state, or could require the user to authorize an update explicitly before it signs the new hardware and reprogramms itself. This architecture allows a manufacturer to commit to a security policy, and force themselves (and would-be adversaries) to follow these.

2.3.2 Threat Model Overview

The adversary in our model is someone who desires to modify the secure hardware implemented in the FPGA or to read back state of the secure hardware implemented in the FPGA. Our work seeks to solve the problem of trusting an external party with maintaining keys that protect the FPGA configuration/state from this adversary. This requires a distinction between trust in operational processes and trust in functionality. In particular, we assume that the FPGA manufacturer is trustworthy at the time the device is created and provisioned, but that the manufacturer may become untrusthworty at a later time, either by being compromised, legally compelled, or having shifting business priorities. Thus, we assume that the original functionality of the FPGA as initially provisioned contains no backdoors or other malicious components, but that any long-term keys maintained by the manufacturer can be compromised.

We ignore the threat of implementation bugs in the secure hardware application, and side-channels on the FPGA that may inadvertently compromise the security of the system [269]. Though likely to exist, we stress two points: first, existing secure hardware also suffers implementation bugs and side channel attacks, and second, our architecture is better able to handle these problems by allowing comprehensive updates.

2.3.3 Motivating Example

As a motivating example of customized secure hardware, we will focus on a secure co-processor with remote attestation. While there are other applications that can be built using our design, secure co-processors are a powerful example that enables a wide range of security applications.

Intel Software Guard Extensions (SGX) [63, 64] is an extension introduced by Intel to their CPUs which provides a **Trusted Execution Environment** (TEE), allowing developers to write software that executes in a context isolated from the rest of the system, including the operating system. SGX also supports remote attestation of the software running in this TEE, but is designed to only allow Intel to verify remote attestations. Others that use SGX for remote attestation must

trust Intel to verify that a remote system is running the code it claims to be running.

In our motivating example, say a company needs SGX-like capabilities, but wishes to use a different party (or even itself) as the trusted source which provides the proof and verification needed in the remote attestation process. This is not possible with Intel (or any existing systems today), so this company would use or design a secure co-processor targeted at an FPGA that provides a TEE with remote attestation. When combined with our self-provisioning system with updates, they can trust that an adversary will not be able to alter their design and, by extension, trust that their TEE will behave as they designed.

This company also wishes to be able to respond to vulnerabilities and deploy patches to their secure co-processor. This comes from experience, as there are numerous examples of vulnerabilities discovered in secure hardware after its release [179, 379, 438, 431, 430, 100, 134, 135]. With the ability to update, the company protects itself from being locked into a vulnerable system or needing to recall physical hardware. Updates, they determine, should be signed by them and should also be verified by their users through the use of a PIN provided in a separate (assumed secure) channel to the user. In Section 2.7 we will discuss our implementation of this specific co-processor system.

2.4 Self-Provisioning

The goal of this work is to ensure that we can program an FPGA with a configuration implementing some custom secure hardware and trust that a malicious party cannot modify it. On the surface, secure boot would appear suitable for this. A secure boot system operates by verifying a signature over a booted configuration against a public key programmed into the system's configuration, such as a secure storage device. The trusted developer has the corresponding secret key and is theoretically the only party that can generate a correctly signed configuration. However, if this secret key is leaked to another party, then this party can put any configuration into the device.

Our solution still makes use of the IP protection hardware used by prior work [434, 352], but changes how the secure boot keys are managed. The problems with the use of secure boot are not

related to how the hardware is implemented – the IP protection hardware was never compromised. It is the business processes that are used to protect the keys that we eliminate. Our self-provisioning system achieves this by generating the secure boot key pair on the device and storing the secret key in the device's storage. The system uses this key to sign a single initial configuration, which then becomes the only configuration that can exist in the FPGA.

The self-provisioning system is simply a trusted piece of software that is run on the device itself to generate keys which will be stored on the device and never exposed.

First, the FPGA is empty with no secure boot set up. The **self-provisioner** configuration is loaded and executes a series of steps, as summarized below:

(1) Generate a keypair for the secure boot system.

(2) Sign the **initial** FPGA configuration with the generated secret key.

(3) Store the secret key in secure storage.

(4) Program the public key to the secure boot system on the device.

At this point, the FPGA's secure boot has been set up and the keys are stored in secure storage on the device. Only the single configuration, determined at provisioning time is allowed to be loaded as it is the only one which has been signed by the secure boot keys. A power cycle of the device will then cause this **initial** configuration to be loaded onto the FPGA. In order for a different configuration to be loaded, it must be signed by the secret that only exists on the device and must be authorized by the security policy of the update mechanism (discussed in the next subsection) of this initial configuration.

The **initial** configuration could be the desired secure hardware application itself (**e.g.,** the secure co-processor with remote attestation), if known at provisioning time. If unknown, or if more flexibility is desired, an option would be to load an initial configuration that does not have any secure hardware application, but can have an update policy that suits the protection desired until loaded with the initial application (**e.g.,** a one-time use key). The update system would then be used to load the actual secure hardware application onto the FPGA. Note that this will result in overwriting the update system's policy with that of the secure hardware application's policy.

2.5 Policy Controlled Secure Updates

The secure update system provides the second component of our platform that allows for applications to make use of the FPGA's reprogrammability. As described in the previous section, once self-provisioning is complete, only a single configuration can exist in the FPGA. However, since the generated secret is accessible to the FPGA, the FPGA can authorize a new configuration. Therefore, to allow for updates, a subsystem needs to be implemented by developers that will implement a security policy. This subsystem will receive updates and will verify that they conform to the selected security policy before using the secret key to authorize an update.

The update subsystem will enforce a security policy, but this policy must be selected and implemented by the developer of the application. Examples of security policies are:

- Update signed by a trusted developer.
- Correct PIN input by user at update time.
- User PIN and trusted signature required.
- No updates allowed.

This list of policies is not exhaustive, but is representative of potential policies. What this enables is choice for the secure hardware developer. They could trust their own processes (to safeguard keys), or, better yet, safeguard against leaks by utilizing a policy which requires signing **and** a PIN, and perhaps extend the policy to allow a new key for signing updates to be regenerated through some local action.

To support this, we require the developer to implement the enforcement of the chosen policy as part their application. This is because these implementations depend heavily on the capabilities of the device and developers will have their own requirements, such as signature algorithms or input devices, that cannot be prescribed for all use cases. We give an example implementation that is not portable outside of our device used for implementation in the next section, but can be used as an example to build other update systems off of, even when implementing a different security policy.

In general, the secure update system is responsible for performing two tasks, irrespective of

the implementation or chosen policy. The first task is to receive updates and enforce that these updates adhere to the security policy before allowing them to be authorized (such as verifying a signature or user PIN). The second task is to use the device-only secret key to sign updates that pass verification and program the signed update to the device. Therefore, an update subsystem must perform these steps:

(1) Receive an update.

(2) Verify that the update conforms to the update security policy.

(3) Use the secret key to sign the update.

(4) Overwrite the existing FPGA configuration such that the update will execute in future power cycles of the device.

As the update system is implemented as part of the initial configuration of the FPGA that is authorized by the self-provisioning system, there is no other way to change the configuration. Therefore, the configuration is secure from being overwritten except by another update that conforms to the chosen policy. This requires that the developer implements the update policy correctly, as there are several attacks, such as man-in-the-middle, downgrade and rollback attacks, that can compromise a security policy that performs only simple authentication. Therefore, update best-practices should be followed, such as the use of sequence numbers and signatures, when implementing a security policy. This is further discussed in the next section, where we discuss which attacks that the update policy we implemented defends against and which it is still vulnerable to.

2.6 Implementation

To demonstrate our platform, we implemented a self-provisioning system and an example application that includes an update subsystem. Our example application is a secure coprocessor that offers similar features to SGX, and is described further in the next section. In this section, we present how we implemented the self-provisioning system and the update subsystem, which any implementation of our platform will need to provide. We also describe the implementation of a secure storage capability in our device, as both the self-provisioning system and the update system

require a secure storage system. We implemented our demonstration application using the Xilinx ZCU102 Evaluation Kit. This system combines a quad-core ARM CPU and a Xilinx FPGA and includes all of the needed IP protection hardware that is required for our platform.

2.6.1 Self-Provisioning

In an ideal system, the FPGA would have direct internal control over the IP protection hardware, with all other peripherals restricted from accessing these systems. However, we were limited by the device we used for our implementation, in that the FPGA does not have direct access to most peripherals in the device's interconnect design. Instead, the coupled ARM CPU is the master of the system, meaning that our provisioning system needed to be run as a software program rather than as a system in the FPGA. This imposes some increased risk of exposure of generated keys, as the ARM system memory is more accessible than the FPGA, but since the self-provisioning system is expected to execute in a trusted facility, this increased risk can be mitigated.

The self-provisioning system that we implemented performs the tasks outlined in the previous section. The provisioner (**e.g.,** the device manufacturer or distributor) will load the self-provisioner onto the device's persistent storage (in our case, an SD card) along with the initial FPGA configuration to be signed. We, acting as the provisioner, have generated the self-provisioning operating system using Xilinx's proprietary tools such that when the device is powered on, the provisioner is executed.

Once booted, the provisioner loads a simple Ubuntu filesystem that executes a single script. This script generates an RSA-4096 keypair for the secure boot system (the ZCU102 secure boot hardware uses 4096-bit RSA keys) and stores it securely. As the only persistent storage available on our device is the SD card, we also leverage additional IP protection hardware that is used for FPGA encryption. This hardware utilizes a small amount of secure storage (battery-backed RAM (BBRAM)) that cannot be read once it is programmed. The self-provisioning system generates an encryption key, programs the encryption key to the BBRAM, and uses the encryption key to encrypt the generated secure boot keypair. On each future boot, the encryption hardware can

decrypt the secret key if needed without it being decryptable outside of the device.

Once the keypair has been generated and the secure storage initialized with the encryption hardware, the self-provisioner uses the keypair and Xilinx's tools to generate a signed boot image containing the initial FPGA configuration that is in the proprietary format used by our device. The output file is then placed onto the SD card so that it will be loaded on the next power cycle of the device. Finally, the self-provisioner will program the generated public key into the IP protection secure boot system of the device, locking the device to only being able to run the boot image that was generated, which contains the initial FPGA configuration.

At this point, the self-provisioner is finished and reboots the device. On the next boot, the signed FPGA configuration will be running and will be the only hardware that can be loaded into the FPGA, as the secure boot system will not let any other configurations that are not signed by the key into the FPGA, and no other such configurations can exist, since the secure boot key only exists on the device itself.

2.6.2 Update System

As required by our platform's architecture, the self-provisioning system locks down our device so that only a single FPGA configuration can exist in the FPGA. To support updates, our platform requires that developers include an update subsystem that will implement a security policy, but we require that the developers provide their own implementations. This is because developers need to make application-specific and device-specific decisions about how to implement the system. In this section, we describe the implementation we used for our application that demonstrates what these application-specific and device-specific can be.

The update system that we provided implements the required functionality of our platform. We selected a security policy that requires a trusted signature over the update and the input of a user's PIN before the update will be accepted. The verification of the security policy is performed by the FPGA, but because the FPGA does not have direct access to the SD card on our device, and because the boot image format that the update must be converted to is also proprietary,

the actual generation of the boot image cannot be done in the FPGA. Instead, when the FPGA authorizes an update, the device will reboot into a simple update operating system that is similar to the self-provisioning system previously described. This means that our update operating system is implemented partially in the authorized FPGA configuration, but also in the update operating system and a trusted bootloader.

When an update is authorized, the update subsystem will store a flag into the secure storage that is only accessible to the FPGA. Upon reboot, the bootloader will check for the existence of this flag and boot into a different operating system. This update system in the FPGA will then release the private key to the operating system after the trusted bootloader indicates that it has booted. The update operating system's only task is to use the secret key and Xilinx's tools to generate a compatible boot image that contains the updated FPGA configuration. Once it has generated this boot image and placed it into persistent storage, the operating system will reboot the device into normal operation.

As can be seen, our device has several limitations that require special implementation considerations, specifically the fact that the FPGA does not have direct access to most system peripherals. In addition, for the enforcement of our security policy, we require user PINs to be six digits in length and we require all updates to be signed using the ED25519 signature algorithm. Other update systems may choose to use different requirements. We also make use of the MicroBlaze soft CPU to implement the update system, whereas other implementations may choose to use other methods, such as pure Verilog or a different CPU. Because of these considerations, we do not provide a single implementation, as any implementation depends upon the capabilities of the device, the requirements of the application, and the exact update security policy that is chosen.

2.6.3 Secure Storage

As mentioned in the previous two sections, the self-provisioning system and the update system both need to store secrets that are only accessible to the FPGA. However, our device does not provide such a capability directly, nor does it allow for the FPGA to directly write to the SD card.

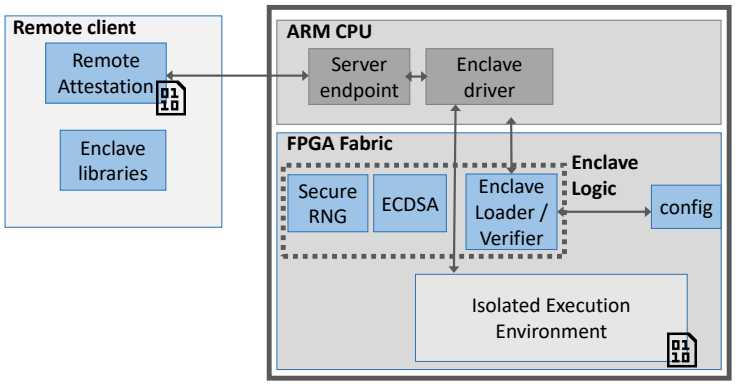


Figure 2.3: Secure Coprocessor and Remote Attestation Design: Here we run the FPGA as a coprocessor and are able to enforce isolation and perform remote attestation. A remote attestation client uploads a program to an untrusted server. The program is launched in a Isolated Execution Environment in the FPGA by enclave logic, which also signs the program code and performs a key exchange. The driver communicates with the program in the enclave over a shared buffer and relays data to the client.

To solve this problem, we leverage the built-in encryption hardware, as mentioned previously, in the form of an AES accelerator that is backed by a secure encryption key storage in BBRAM. The self-provisioning system initializes this accelerator with a random key that never is stored except in the BBRAM and uses the accelerator to encrypt data. Using the accelerator, we can achieve a secure storage that prevents data from decrypted outside of the device.

However, the FPGA cannot directly pass data to the AES accelerator. Instead, we require that a proxy be run in the CPU of our device that passes data between the FPGA and the AES accelerator, and stores the encrypted data onto the SD card. To further protect the data, we have also implemented a corresponding subsystem in our application that interacts with this agent, which encrypts any arbitrary data generated by our application using an FPGA-only key that is stored in a dedicated eFuse array only accessible to the FPGA. This ensures that when passing data to the CPU agent after boot that no cleartext data is available in the CPU's memory.

2.7 A Customized Secure Coprocessor with Remote Attestation

In Section 5.3 we described a motivating example where a company wishes to have a secure co-processor with remote attestation where the root of trust is flexible (**i.e.,** not the manufacturer,

as in SGX). In this section we elaborate on the hardware design, the software development kit to develop software applications, and two example software applications (password manager and contact matching) that were built with our software development kit.

2.7.1 Hardware Design

2.7.1.1 Isolated Execution Environment

The code that can be provided to the secure co-processor to run in an isolated manner is in the form of a partial configuration bitstream. There are two options we support for the internal architecture of this hardware. The surrounding logic is identical in both cases, but it is the contents of the configuration bitstream which differ.

Option 1: Software Enclave.

To provide a software environment for software isolation and remote attestation, we implemented a MicroBlaze [78] soft CPU inside the FPGA as part of the secure hardware application. Any code that executes in this CPU is isolated from the untrusted operating system and can be trusted to execute once loaded. Developers provide their code to the SDK, which will then generate the needed logic to execute this code in a MicroBlaze CPU.

Option 2: Hardware Enclave.

Alternatively, developers can directly provide hardware, so long as it is able to perform the interaction with the untrusted software. This does not imply the developer has to develop hardware. They can develop logic directly for the FPGA in any manner that they choose, including by synthesizing the developer's software (C code) into a compatible bitstream using high-level synthesis, as is described in Section 2.7.2.

The developer can make the decision between having their enclave's code (provided as C code) synthesized to hardware or executed on a soft CPU based on the complexity of the application – more complex applications are more difficult to synthesize to hardware, but an application synthesized to hardware will have better performance. The SDK will generate a resulting partial

bitstream based on the developer's choice and the synthesis results that either includes the application directly implemented as FPGA logic, or a soft CPU in the FPGA logic that executes their application's code. The SDK also generates an untrusted program (**i.e.,** the "Enclave driver") that runs on the device's (untrusted) CPU to interact with the enclave program via a memory buffer in the FPGA.

2.7.1.2 Enclave Code Loader

In order to securely program this co-processor, we utilize custom logic that ensures that when any trusted code (**i.e.,** a trusted "enclave" program, similar to SGX) is loaded, a hash of this program is taken and a signature verification are performed. As illustrated in Figure 2.3, the code of the application is provided to the logic in the form of a partial bitstream, which specifies a configuration which will reprogram only part of the FPGA. The enclave logic will use the internal configuration access port (ICAP) to program the partial bitstream (the enclave program) into the area of the FPGA reserved for executing the secure enclave, leaving the rest of the FPGA (**e.g.,** enclave logic) untouched.

In addition, the enclave logic reads an ECDSA private key from the secure storage, and uses it to sign the hash of the bitstream and a message from the enclave during the remote attestation process. As shown in Figure 2.3, a remote client can upload a program to services running in the untrusted operating system, which will then pass the program to the enclave logic.

2.7.1.3 Remote Attestation

The attestation protocol implemented by our secure hardware and companion software is shown in Figure 2.4. In this protocol, a remote verifier uploads a program (in the form of a partial bitsream) signed by its Ed25519 private key (SK_v) [172]). The program will be launched by the enclave logic, and the verifier will be notified upon completion. The verifier will then request an attestation by uploading its signed public key (PK_v). The enclave logic then generates an ephemeral key pair for this attestation to establish a shared secret for the enclave ($PK_{enclave}$, $SK_{enclave}$), and

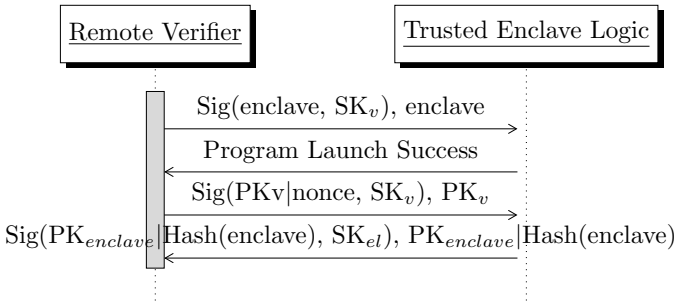


Figure 2.4: Remote Attestation Sequence: In the remote attestation protocol, the remote verifier uploads a program (enclave) signed by its private key (SK_v). The enclave launches the program and notifies the verifier, which then requests an attestation by sending its signed public key (PK_v). The enclave logic uses this key to derive a shared secret for the enclave and responds with a signature of an ephemeral public key for the enclave ($PK_{enclave}$) and the hash of the enclave, signed by a long-term key for the enclave logic (SK_{el}).

signs $PK_{enclave}$ and the hash of the enclave program with its long-term attestation key (PK_{el}, SK_{el}). The enclave sends these to the verifier, along with a certificate chain configured at provision time by the root of trust for this device. Using this certificate, the verifier then verifies the signature and checks that the hash matches the expected hash of the uploaded enclave program. If so, the verifier can calculate a shared secret using $PK_{enclave}$ and SK_v, just as the enclave logic calculates a shared secret using PK_v and $SK_{enclave}$. Using this shared secret known only to the verifier and the isolated enclave, a secure channel can be established.

To generate secure ephemeral keys during this process, we have included a cryptographic random number generator within the trusted hardware of the FPGA, as implemented by the Cryptech OpenHSM project [384]. The module draws randomness from both the LSB of A/D conversion noise as well as a ring of digital oscillators implemented as a set of adders with the carry out inverted and fed back as carry in. This entropy is collected and hashed using SHA512 to whiten it. The resulting digest is used to seed a ChaCha stream cipher's key and IV which is used as a PRNG to provide random numbers to the enclave logic to securely generate keypairs.

2.7.2 SDK

In addition to designing the hardware of our software isolation system, we have also designed a software development kit to make it easier to develop software applications that run in the

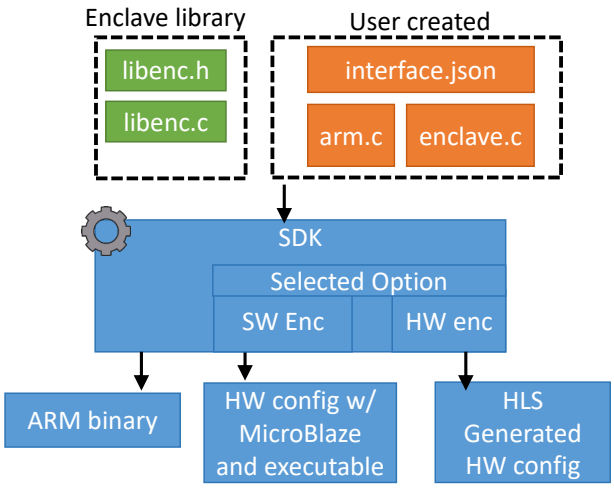


Figure 2.5: SDK Development Flow

system. Figure 2.5 shows the major components of the SDK. A developer creates untrusted code that runs on the ARM CPU of our system in the untrusted operating system (arm.c), code that implements the trusted functions that are run in the isolated enclave (enclave.c), and a description of the API the application wishes to use to communicate between the trusted and untrusted code (interface.json). This interface describes the inputs and outputs of the trusted code as well as the function signatures of the specific methods. The developer also has access to the enclave library (libenc.h, libenc.c) that provides functions to launch an enclave, which is done by interacting with the enclave logic.

The developer provides their code to the SDK. For a software enclave, the SDK will output a partial configuration bitstream (which was pre-built) that contains a MicroBlaze [78] soft CPU (**i.e.,** a processor implemented in the FPGA logic). The SDK will cross-compile the enclave code and add the memory to the configuration bitstream. For a hardware enclave, the SDK will utilize the Vivado [115] high-level sythesis tool, which generates Verilog from C code. Then it will synthesize that design and generate a partial configuration bitstream.

In both cases, the SDK will use the API interface definition to generate communication code between the enclave and the ARM CPU using the dedicated shared buffer. Also, in both cases, the SDK will cross-compile the application code for the ARM instruction set. The (untrusted) ARM

binary's will load the trusted code into the enclave using the enclave library.

2.7.3 Password Manager Application

As an illustration of running isolated software in this secure hardware module, we implemented a password manager that encrypts stored credentials under a master password. Passwords are encrypted and decrypted in an enclave with only the encrypted data being stored in persistent storage. To access a password, the enclave must be provided the encrypted data and a master password. The enclave then derives a decryption key using this password and a device-only key that can only be accessed from the enclave.

To use the manager, a user provides their master password to a client program which interacts with the enclave. The user then has the option to enter information for passwords, usernames and identifiers (**e.g.,** a website). This information is given to the enclave to encrypt, and passed back to the client application to store in persistent storage. Retrieving data is achieved by interacting with the client program and requesting data by its identifier, which will cause the enclave to decrypt it and return it to the client. This password manager is similar in design to an example application SGX provided by Intel [66].

Our implementation cannot remove all possible attack vectors, as the password manager must still function to provide data in plaintext in order for it to be useful for users to interact with unmodified programs. However, we can force any attacks to be **online**, in the sense that the adversary must query the password manager in the trusted enclave, rather than simply be able to make copies and reveal the entire database. This is because the encrypted password database can only be decrypted using the user's master password and the FPGA's device-only key. Even if the database is exported and the user's password is compromised, the data cannot be decrypted without interacting with the enclave running on the device on which it was first encrypted. We present a performance analysis of user interaction with the password manager in Section 5.6.

2.7.4 Contact Matching Application

As a second example to show how our isolated environment can execute code that has been synthesized into FPGA logic using high-level synthesis, we have developed a second application. This application emulates the SGX-enabled contact discovery service operated by Signal [312], except implemented using C++ and synthesized into hardware using our SDK. This application's purpose is to receive an encrypted list of contacts (**i.e.,** phone numbers) from a user and determine the intersection of this with a database of all registered users of the service. The solution used by Signal is designed to prevent the operators of the service from learning the contacts in the uploaded list while still allowing for users to determine the intersection with the total database. By executing in an SGX enclave, Signal is able to conceal which contacts are found to match, and return an encrypted result to the user. Our contact matching application provides similar functionality, but executes its code in FPGA logic that has been synthesized using our SDK. We present the performance of this application in Section 5.6.

2.8 Evaluation

As an example secure hardware application, we built a secure co-processor with remote attestations. Here, we we evaluate the performance of example applications for this secure co-processor along with associated metrics about how long it takes to load and perform a remote attestation. For all of our applications we continue to use the ZCU102 Evaluation Kit running Ubuntu 15.10.

2.8.1 Software Enclave Performance Benchmarks

To test the performance impact of executing code on a Microblaze CPU, we designed several microbenchmarks to test memory and computation performance, along with end-to-end performance.

Software Enclave SHA512 Performance We created a program that hashes a buffer of random data using SHA512 in both an enclave and directly on the main CPU. As the enclave executes on the embedded Microblaze CPU, we expect the performance to be much worse, and this experiment is intended to determine if using our SDK to create enclave programs imposes additional overhead.

The performance of the Microblaze enclave is approximately 20x worse than the reference implementation on the ARM CPU. However, both implementations scale linearly with the size of the data being hashed. There does not appear to be any overhead caused by using our SDK to develop a program for the enclave, and it appears that the execution performance of the Microblaze CPU is the main performance bottleneck, as expected. We stress that while our system has significantly less performance than that of pure hardware implementations, very few secure applications require the full performance of the main processor, but instead emphasize security, isolation, and ease of implementation over raw throughput.

Password Manager Performance Illustrating the point that the performance impact of our implementation commonly would impact a relatively small fraction of the overall perceived performance, we measured the time to add and retrieve passwords from the password manager application described in Section 2.7.3, for passwords of up to 100 characters in length. As seen in Figure 2.7, both with and without running in an enclave results in an average 202ms latency (with less than 0.3 difference in the worst case). Likewise, for reasonable passwords up to 100 characters, the latency for decrypting a password from the manager is roughly 120ms for both implementations, well within the realm of usability (for passwords much larger than that, the impact of the performance difference does become noticeable as more time is spent in the enclave).

Enclave Memory Access Performance To measure the memory access performance of an enclave, the enclave is simply tasked with copying an input buffer to an output buffer, and the performance is compared to the ARM CPU's performance at the same task. We measured an overhead for Microblaze access times ranging linearly from 100x for small chunks of data (0-250 bytes) to 12x for larger chunks (2 Kbytes and larger).

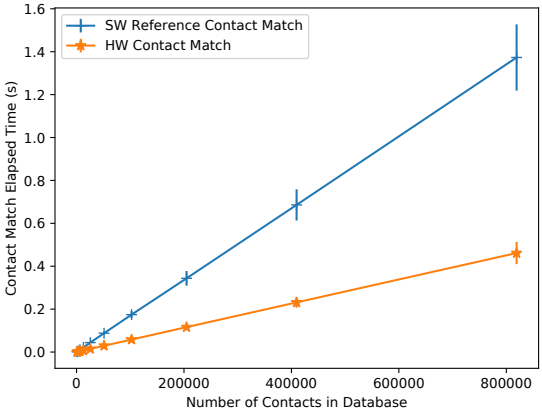


Figure 2.6: Contact Matcher Performance: Performance of matching a contact list against a larger database in a software-only implementation and an HLS-synthesized version. The hardware version achieves an average of approximately 3x compared to the software version.

2.8.2 Hardware Enclave Performance

To show that our SDK can also achieve acceptable performance for large scale processing, particularly through high-level synthesis (compiling C code directly to hardware), we developed a second application that performs a similar service as the contact discovery service operated by Signal. As discussed previously, the purpose of our application is to receive a list of phone numbers from a user and determine the intersection with a larger database, and then return the result to the user. We compared the performance of this application to a software-only implementation that used the same contact list and database. As shown in Figure 2.6, the synthesized hardware version achieves a throughout of up to 3x compared to the software solution. We used contact list sizes of 128 contacts, represented as SHA512 hashses, and database sizes ranging from 800 contacts to 819,200 contacts, also represented as SHA512 hashes.

2.8.3 Enclave Logic Microbenchmarks

Enclave Loading Performance Our final benchmark measures the throughput of loading enclave program binaries of various sizes. After testing using binaries ranging in size from 20 KB to 1 MB, the throughput remained constant at 35 KB/s.

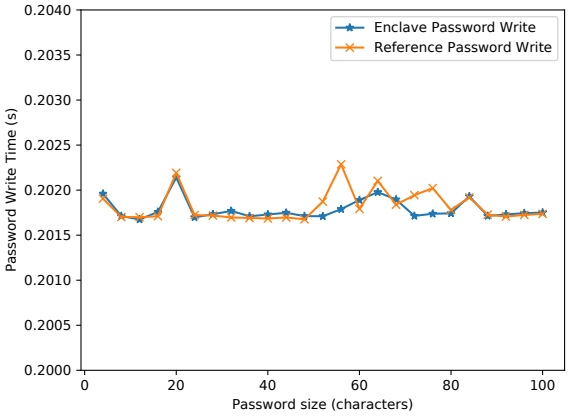


Figure 2.7: Password Manager Write Performance: Time spent adding passwords to the password manager when protected by an enclave and when using a reference implementation running completely on the ARM CPU without an enclave.

Remote Attestation Performance To measure the end-to-end performance of performing a remote attestation, we implemented a private set intersection calculation program that calculates the intersection of two sets of integers in an enclave, with one set being uploaded in encrypted form using the shared secret negotiated by the remote attestation protocol, and the other provided to the enclave by the local host, similar the contact discovery feature used by Signal [312]. In each attestation, a fixed amount of data is passed in each message, which is the public key of the verifier in one message, and then the signed public key and hash of the enclave in the response. This experiment measures the average time to pass these messages, for the enclave logic to generate the keys and sign the message, and the time for the client to verify the response and calculate the shared secret. After performing 1000 trials in ideal laboratory network conditions between a verifier and the device running the trusted enclave logic, the average remote attestation time was 107.2 ms with a standard deviation of 8.604 ms.

2.9 Discussion: Ideal Hardware Support

In our design, we used commodity FPGA hardware, but there may be additional fixed hardware that could simplify or improve support of flexible secure hardware. Here, we explore subtle

architectural modifications to the fixed hardware in FPGAs that could improve or simplify our implementation.

Dedicated FPGA storage In our implementation, we used a combination of BBRAM and a small kernel of trusted software to load a key into the FPGA, allowing it to later encrypt writes to and decrypt reads from a system storage without needing to trust the CPU. A more elegant solution could allow the FPGA to directly write to its own persistent storage that is not accessible from the CPU.

Reprogrammable Secure Boot Most existing secure boot systems, especially as are used in FPGAs for IP protection, are essentially one-time programmable. This means that once provisioned, it is difficult to re-provision a device to a new owner, and virtually impossible to completely remove prior state. A partial re-provisioning is possible with the cooperation of the previous logic under our system, as the secure boot keys can be provided to a new provisioning step, but there is no way to force this. If manufacturers were to implement more complex secure boot systems that could be reset under certain circumstances, such as only by an internal request that was booted by the secure boot system, then we could have more comprehensive re-provisioning options.

FPGA Secure Boot Our secure boot only supported booting trusted code into the main CPU, which in turn could program the FPGA. This required a small amount of trusted code that would program the FPGA, and then remove the CPU's access to reprogram or introspect on the FPGA before booting the untrusted OS on the CPU. An alternative more elegant design however, could allow the secure boot to directly program the FPGA, obviating the need for any trusted code to run on the CPU.

FPGA control of CPU As a further extension, the FPGA could have control over the CPU, rather than vice versa. For example, the FPGA could be given control over the TLB, cache, and ring level of the CPU, allowing it to halt the CPU and decide what code it should be running and from what permission level. This would allow the FPGA to take advantage of the full power of the CPU, running enclave code on it while keeping it isolated from the untrusted operating system,

and clearing caches or encrypting memory before swapping the untrusted OS back in.

2.10 Conclusions

In this chapter we introduced a new mechanism that enables developers to optimize their applications' security by letting them define their own secure hardware features. We use FPGAs to enable developers to implement customized secure hardware without depending on human / business processes to provide updates, bug fixes, and maintain the secrecy of keys used to protect the FPGAs configuration process, We introduced the concept of self-provisioning and a secure update process which allows for policies which govern whether an update is allowed or not. As a proof of concept, we implemented the framework on the Xilinx Zynq Ultrascale+ FPGA and built a secure co-processor with remote attestation that has a flexible root of trust.

Chapter 3

Software Packet-Level Network Analytics at Cloud Scale

Next, we look at the current state of network telemetry and monitoring in the cloud. We highlight the lack of flexibility, performance, and efficiency in today's network monitoring solutions. We then propose a new, highly performance, flexible, and scalable network telemetry and monitoring platform that allows developers to optimize their network monitoring applications at a per-packet granularity.

As networks grow in speed, scale, and complexity, operating them reliably requires continuous monitoring and increasingly sophisticated analytics. Because of these requirements, the platforms that support analytics in cloud-scale networks face demands for both higher throughput (to keep up with high packet rates) and increased generality and programmability (to cover a wider range of applications). Recent proposals have worked toward these goals by offloading analytics application logic to line-rate programmable data plane hardware, as scaling existing software analytics platforms is prohibitively expensive. The rigid design and constrained resources of data plane devices, however, fundamentally limit the types of analysis and the number of tasks that can run concurrently. In this chapter, we demonstrate that generality need not be sacrificed for high performance. Rather than offloading entire analytics applications to hardware, the core idea of our work is to offload only critical preprocessing tasks that are shared among applications (e.g., load balancing) to a line-rate hardware frontend while optimizing the core analytics software to exploit properties of network analytics workloads. Based on this design, we present Jetstream, a hybrid platform for network analytics that can run custom software-based analytics pipelines at through-

puts of up to 250 million packets per second on a 16-core commodity server. Jetstream makes sophisticated, network-wide packet analytics feasible without compromising on generality or performance, enabling developers to optimize their applications' network security, network monitoring performance, and network monitoring efficiency.[1]

3.1 Introduction

Effective network management requires traffic analytics: the capability to mine critical information from packet streams, which can be used to trigger actions in the network or guide subsequent decisions. Traffic analytics is a core component in today's reliable networked systems that is used to help meet stringent security [407], correctness [256, 297], and performance guarantees [143, 170]. Historically, we largely relied on humans in a network operation center to watch some transformed version of the data (e.g., graphs) and manually interpret the data to then take action. This approach does not scale to today's data center or wide area networks which continue to grow in complexity, size, and traffic. Instead, today, the ability to continuously perform automated and sophisticated analytics across the entire infrastructure is imperative [364, 255].

Given the importance of the problem, in recent years, many novel and compelling architectures and systems for fine-grained network monitoring in cloud-scale environments have been presented [256, 332, 250, 442, 456]. At the core of each proposed system is an underlying processing engine that analyzes raw data. The design of such a processing system is the focus of our work.

In an ideal world, the system would enable arbitrary, sophisticated analytics that consider every single packet traversing a network. A network operator should be able to write multiple custom analytics applications to run in parallel. These applications can be interactive queries or long-running, continuous analyses over a network packet data stream.

In a nutshell, the analytics system must be **general** to enable arbitrary and runtime-configurable applications through a programmable interface. Equally important is high **performance** to allow for economically feasible network-wide coverage and parallel analytics applications. This ideal of

[1] Work published in IEEE TNSM 2021 [324]

general, software-based analytics on every packet in a cloud-scale network is expensive to realize. Consequently, there has been a long history of work compromising on various dimensions with the goal of making this vision practical.

Historically, flow aggregation and sampling (e.g., with IPFIX [197]) have been the main tools of network operators to reduce the amount of information to analyze. Both approaches are appealing because they can be practically implemented in resource-constrained hardware switches. Aggregated network records, however, hinder fine-grained analytics that are required for a wide range of performance- and security monitoring use cases [332, 456], while sampling compromises on data fidelity and accuracy [220, 225, 362]. These limitations motivated researchers to propose custom algorithms and probabilistic data structures (e.g., sketches) that provide provable accuracy and can be implemented in hardware [291, 443, 305]. Still, sketching only supports basic statistical analysis, limiting generality. For example, more intricate analytics logic such as detecting a network loop, where a packet traverses the same switch twice, cannot be realized using sketches.

These compromises on generality and data granularity are increasingly problematic today, as there is a growing number of applications that need to perform analytics on data from every single packet in a network, for example for machine learning in intrusion detection or traffic classification systems [314, 327, 349, 416, 206, 159, 436, 375]. For these tasks, analysis is sophisticated and application-specific, and hence impractical to implement as a sketch or in hardware. To accommodate such applications, the community presented ways to analyze entire traces of packets in software. Performance limitations, however, meant that these proposals suffered from poor visibility, e.g., limited to a single switch [364] or a specific class of flows, which again makes them unsuitable for the many modern analytics applications mentioned above.

Today, we are left with two directions that research has taken toward the goal of being able to analyze every packet in a network for a wide range of applications. The first direction is to compile analytics tasks to run on modern programmable switches [332, 250] (see left side of Figure 3.1). This is challenging as hardware resources on these switches are heavily constrained. To illustrate this, we compiled the Sonata [250] queries available [101] to the Intel Tofino programmable forwarding

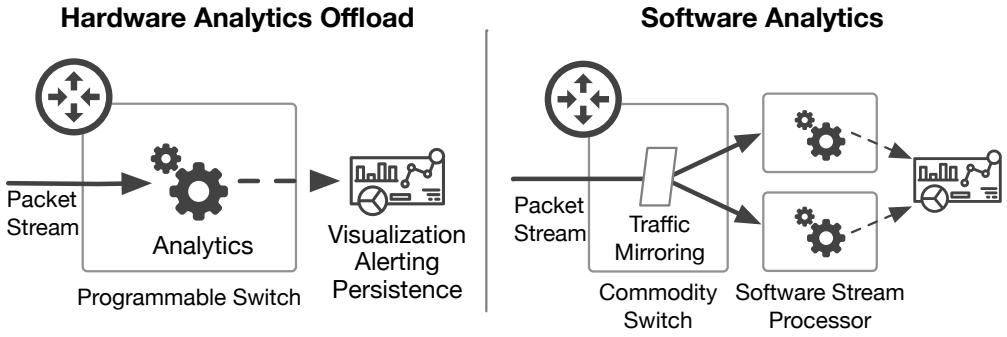


Figure 3.1: Previous Network Analytics System Architectures

engine (PFE) using two levels of refinement. Only two of the seven queries fit within the resource limits of the chip (see Section 3.8.2). This leaves the other queries as not currently being practical and raises questions about the feasibility of enabling multiple queries to run simultaneously.

The second direction is to adapt a pure software architecture for network analytics, using a map-reduce-style, scale-out system such as dShark [442] (see right side of Figure 3.1). While this allows for horizontal scalability and supporting multiple queries simultaneously, performance is still a significant challenge. In an end-to-end performance evaluation, dShark's throughput is 10.6 million packets per second on a 16-core server. This would result in needing to dedicate 96 servers to monitor a single cluster in a modern data center [376] for a single application (see more in Section 3.8.3).

In this chapter, we introduce a third direction that balances the two previously presented extremes. Our proposed system, Jetstream, uses a hardware-software co-design and can efficiently analyze hundreds of millions of packets per second for multiple simultaneous applications allowing for network-wide, packet-level analytics without compromises. Our design is based on two key strategies.

First, we leverage programmable switches for **system-level** offload: Rather than pushing down entire analytics applications to programmable data plane hardware (i.e., compiling a query to P4 [177]), Jetstream offloads system-level tasks that are necessary for **all** analytics applications to a programmable data plane frontend. For example, tasks such as extracting packet features,

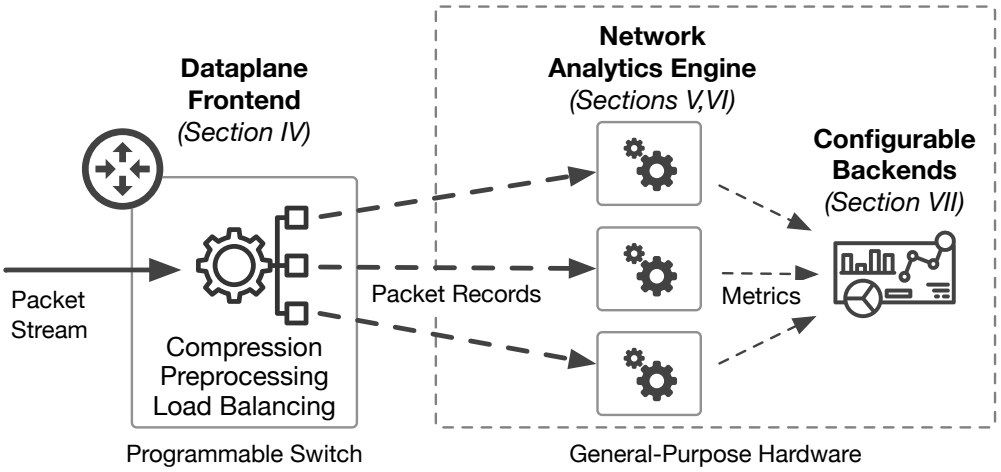


Figure 3.2: Jetstream Architecture Overview

compressing and organizing packet records for efficient processing, and steering data streams can efficiently be implemented in hardware but are expensive to run in software. By offloading them to programmable switches, we eliminate much of a software analytics platform's work without overloading the programmable switch.

Our second strategy is to carefully optimize Jetstream's software component to exploit both the properties of network analytics workloads and our partitioning between hardware and software. For example, the structure of packet flows is inherently suitable for distribution across servers (see Section 3.5.1). Since load balancing is offloaded to the programmable switch frontend, Jetstream's analytics pipelines (which run application-specific logic) can be designed to operate independently of each other. This eliminates resource contention to improve both performance and scalability. Finally, guided by workload characteristics, we apply a series of domain-specific system optimizations. These optimizations allow for significant performance gains over general-purpose systems without impacting application logic or accuracy. The resulting high-level architecture is depicted in Figure 3.2.

We implemented a complete prototype of Jetstream. The data plane frontend runs on a Barefoot Tofino PFE at line rate of 3.2 Tb per second and allows for dynamic adding, removing, and scaling of analytics tasks without reloading the programmable switch. The core software

analytics engine is implemented in C++. It consists of a framework and a library for writing custom analytics pipelines that compute relevant metrics from network packet record input streams. It integrates optimizations that include kernel-bypass input/output, zero-copy message passing, high-throughput concurrent queues, batching, and accelerated hash tables. Lastly, a configurable backend for aggregating and querying metrics provides an interface to network operators or control platforms. We evaluate Jetstream with real-world traffic traces using seven example analytics applications: a heavy-hitter detector, a software load balancing profiler, a Slowloris DoS attack detector, a SSH brute force detector, a SYN flood detector, a TCP sequence analyzer, and a traffic statistics/accounting application.

We published partial results on an early design of the software analytics component of Jetstream in [325]. We now fundamentally extend the earlier processing engine by integrating it with hardware-based telemetry systems and introducing an independent, parallel processing pipeline architecture as a core design strategy. Together with the data plane frontend and a database backend, this article provides an end-to-end system which we evaluate in a realistic multi-server deployment.

Our evaluation shows that individual application throughputs range from 5.4 to 15.9 million packets per second for a single core. Jetstream scales linearly with core count across machines (or between 86.4 and 254.4 million packets per second on a 16-core server). For comparison, using a 16-core server, Spark (Sonata's backend) can handle 1.4 million packets per second and dShark can handle 10.6 million packets per second. A task that would take 24 servers in dShark only requires a single Jetstream server, demonstrating how Jetstream's design and optimizations make the ideal of sophisticated and network-wide analytics practical.

In the remainder of this article, we first motivate the need for Jetstream by discussing the progression of analytics systems towards increasing generality (the ability to support a wider collection of applications) and performance (the ability to handle more traffic) in Section 5.2. We then introduce Jetstream and its architecture in Section 5.3. This architecture consists of three main components which are then detailed: the programmable data plane frontend (Section 3.4), the core software network analytics engine (Sections 3.5 and 3.6), and the on-demand aggregation

and query backend (Section 3.7). We evaluate Jetstream in Section 5.6 and conclude in Section 3.9.

3.2 Motivation

With recent advances in networking technology, such as software-defined networking [317] and programmable data planes [178, 177], and the rapidly increasing scale of networks, there has been a flurry of research toward improving network monitoring and analytics. Each proposed system has moved us closer to the idealized goal of being able to perform general analytics on every packet in a network. The challenge, of course, is doing so in a cost- and resource-efficient manner. This is where each current analytics platform makes tradeoffs. In this section, we motivate the need for and the design of Jetstream by discussing the most relevant prior systems.

3.2.1 Sketching in the Data Plane

Sketching is among the most resource efficient approaches to custom analytics. Sketches leverage probabilistic data structures to compute summary statistics over large input datasets using a sub-linear amount of memory [146]. OpenSketch [443] provides a library of such sketches to be deployed in programmable hardware platforms, while UnivMon [305] introduces a universal streaming scheme, where a generic sketch in hardware preprocesses packet records at high rates and software applications compute application-specific metrics.

While extremely efficient in space requirements, sketches can only support certain classes of statistical functions and aggregate analysis as they lack visibility into individual packets. For example, an analysis task that cannot be represented with a sketch is the detection of packets that traverse the same switch twice, i.e., a network loop. By design, sketches also over- and under-count events and randomly lose information because of hash collisions in the underlying data structure.

In contrast, an analytics application running on top of Jetstream has visibility into every packet and can therefore calculate any statistic with full accuracy.

3.2.2 Packet-level Software Analytics

There is a growing set of analytics tasks (particularly machine-learning intrusion detection and traffic classification systems) that cannot rely on sketching because they need to either analyze fields in each packet or perform sophisticated, application-specific analysis. Examples of the required packet-level data include packet inter-arrival times [314], TCP receive window [349, 159], and TCP flags [349, 159]. Analytics applications use these and other fields to compute: packet lengths statistics [375, 327], packet arrival order [206], and many other advanced and derived metrics (e.g., Fourier transforms of inter-arrival times, flow idle times, mean packet sizes, flow duration, number of TCP data packets) [314, 327, 375, 349, 416, 206, 159, 436].

To support such applications, there have been proposals to process entire traces of packets in software. Planck [364] demonstrated the ability to mirror packets of interest to a management port of a switch which then sends traffic to an attached server for processing in software. Planck has limited scalability and incurs packet loss due to substantial oversubscription of the management port. To reduce the workload, NetSight [256] filters out traffic that is not of interest, using Berkeley packet-filter (BPF) style filters, before application-level processing, while Everflow [456] pushes both filtering and shuffling into data plane hardware. While these systems improve scalability, the heavy reliance on filtering limits their applicability to debugging tasks and increases operator burden, as operators must know what they are looking for a priori.

Finally, distributed measurement frameworks, such as SwitchPointer [417] or Confluo [283] collect features from regular network packets at the network's end hosts and perform lightweight analysis there. This approach lacks visibility into the core of the network and also requires analytics functionality and applications to be deployed on every single host at the network edge. Finally, Confluo applications must follow a rigid programming model limiting its applicability for the above mentioned applications.

In contrast, Jetstream's high throughput enables scaling without filtering, giving visibility into all packets collected from throughout the network. Applications can flexibly extract features

and compute metrics of interest using a general-purpose language and an unrestricted programming model.

3.2.3 Compiled Queries in the Data Plane

With the emergence of programmable forwarding engine technologies (PFE) [178, 177], researchers have sought to use these platforms to solve scalability issues introduced by previous packet-level monitoring systems by compiling some or all of the processing into line rate hardware.

Marple [332] identified a set of fixed operators that can be compiled to a programmable forwarding engine and used to implement parts of a network monitoring query. This approach offers great performance, but not all queries can fit within the resource constraints of a PFE. For those queries, performance is typically bottlenecked by the backend stream processor. Compiled queries are also problematic for other reasons. First, due to limited resources on these devices, only a small number of tasks can run in parallel [273, 402]. Second, reconfiguring data plane programs (i.e., changing the running monitoring query) is disruptive as it incurs device downtimes on the order of tens of seconds [402]. Third, applications are constrained to use the fixed set of operators available in the PFE programming model. While general, some applications [337] require metrics that are too complex for switch hardware to implement [392]. Fourth, deployment is challenging because overall system throughput is highly sensitive to the application, how it is split between hardware and software, and the workload characteristics (e.g., number of flows).

Sonata [250] reduced PFE memory requirements by introducing a method of iterative refinement for the PFE component of a query. This comes with two additional drawbacks, however. First, iterative refinement requires additional costly hardware resources. In our evaluation (see Section 3.8.2), we find that refinement causes only two out of seven applications to be able to fit on the PFE. Second, refinement relaxes the temporal and logical constraints of a query. Events must last longer than a refinement window to be detected, which is on the order of seconds [250]. Further, even long-lived events can be missed because they may fail to match relaxed thresholds in the coarse-grained early stages of a refined query.

In contrast to these systems, Jetstream leverages hardware (switch) offload for preprocessing logic that is expensive in software and common to all applications. All example applications that we later discuss in Section 3.8.1 require feature extraction, record load balancing, and distribution. By offloading this system-level functionality (rather than application-specific tasks), Jetstream can accelerate **all** analytics tasks and scale predictably and efficiently with the number of running applications while eliminating the need to re-load switch logic to run new or additional applications.

3.2.4 General-purpose Software Processing

An alternative approach to programmable data plane acceleration and offload is to optimize software-based analytics. Software platforms can support virtually any application and can be reconfigured without downtime; however, per-core processing rates are generally low, making operation in environments with high packet rates prohibitively expensive.

There are two orthogonal lines of work in this area. First, language-based tools, such as NetQRE [447] compile queries into efficient C++ programs. Second, and more related to Jetstream, are stream processing frameworks designed to run many applications concurrently and at scale, e.g., dShark [442]. While dShark performs much better than general-purpose stream processors (e.g., Spark, used as the backend of Marple [332] and Sonata [250]), its throughput is still low relative to network packet rates. Reported throughput for dShark, for example, is on the order of 10.6 million packets per second for a 16-core server running one application. This is several orders of magnitude lower than typical data center packet rates, effectively requiring racks full of servers just for analytics.

To understand the limitations of existing stream processing systems and build intuition for Jetstream's design, consider Figure 3.3a, which shows the general architecture of a software stream processor used to analyze packets traces from across a network. The figure illustrates three main steps in the analytics process, each of which has a significant bottleneck that Jetstream eliminates.

First, the frontend of the stream processor must handle load balancing and distribution: forwarding a copy of each packet to an instance of each application that needs to analyze it. Given

the high event rates in network analytics, this task of deciding where each packet should go and load balancing across servers and processor cores is expensive in software, and can easily bottleneck the whole system.

Second, in the application-specific analytics pipelines, sequences of operators transform input packet streams into streams of meaningful information (e.g., metrics or alerts). In these pipelines there are many sources of overhead that cumulatively reduce throughput by an order of magnitude. For example, copy and locking operations in inter-operator queues, pointer-chasing in container-based key-value data structures, and serialization overheads in message-passing subsystems. As Section 3.5.1 explains in more depth, for many network analytics tasks frequent message passing and lookup operations are required, making general-purpose stream processor overheads impact network analytics tasks significantly.

Finally, a typical stream processing network analytics application would aggregate results across the instances to output the metric(s) of interest. This requires each worker to send data to a single aggregation node — a third bottleneck.

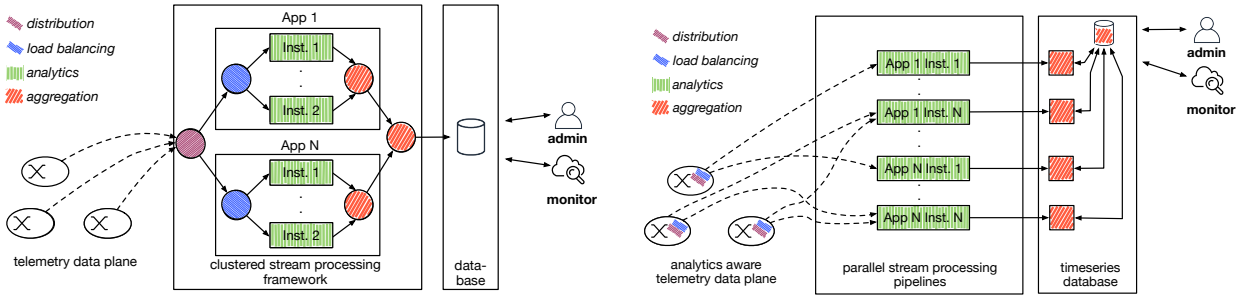
Takeaway. All of these systems have benefits over traditional solutions (e.g., traffic mirroring or flow monitoring), but introduce compromises. Further, while some use programmable network hardware, all still rely, to varying degrees, on software for final metric computation and are therefore subject to the above mentioned bottlenecks. As a result, even for state-of-the-art telemetry systems, Jetstream's capability to support high-throughput and general analytics in software is essential for meeting novel, packet-level monitoring requirements in cloud-scale networks.

Further, software processing as it is possible in Jetstream enables applications written in a general-purpose language and does not limit the complexity of analysis or require sacrificing accuracy to gain performance. Instead, applications can fully leverage the flexibility of general-purpose hardware with ample memory and processing resources to implement complex analytics using, e.g., neural networks, sophisticated stateful logic, or third-party libraries.

As we describe next, Jetstream achieves these goals through a combination of system-level hardware offload and software optimization, which eliminate the bottlenecks outlined above to

enable high-performance network analytics in software.

3.3 Introducing Jetstream



(a) Straw man network analytics system architecture

(b) Jetstream network analytics architecture

Figure 3.3: Telemetry-based network analytics system architectures

Jetstream is a high-performance network analytics system that makes no compromises on generality or performance of analytics tasks. It lets applications perform packet-level analytics, including the calculation of arbitrary metrics, entirely in software and scales linearly with server resources. To overcome the issues observed in Section 5.2, we follow two main design strategies.

First, as Figure 3.3b illustrates, we move distribution and load balancing functionality into network switches. We call this **analytics-aware network telemetry**. We also push aggregation of computed, metric streams to external backend systems. At the core of our system then remain independent stream processing pipelines that are primarily bottlenecked by computational, input/output and data structure overheads. The second design strategy is to optimize these overheads away using a collection of techniques drawn from prior work but adapted for packet analytics workloads.

3.3.1 Using Jetstream

Jetstream is designed to run user-defined applications on records for every packet in a network. These applications are written in a general-purpose language (here C++) and can use a highly optimized set of common network-oriented stream processing operators that are part of the

Jetstream library. In addition to using this standard library, a user can implement operators with entirely customized logic that still benefit from Jetstream's system-level optimizations.

Typical applications implement, for example, header-based intrusion detection [326] or performance monitoring applications, such as a queue depth monitor based on telemetry data from data plane hardware [402]. Common across all applications is the broader goal of network analytics, that is making the vast amount of records exported from network devices comprehensible and useful for the operator. This means that Jetstream applications must perform significant event rate reduction through (application-specific) data aggregation. As a result, the output data of a typical Jetstream application is again a (much lower frequency) event stream of applications-specific data tuples (metrics) for further, sometimes interactive, analysis [447, 323], visualization, network control [170, 234], or archiving [323] in a backend system. As a proof of concept we demonstrate the integration with a time series database system (Prometheus [72]) as one possible backend.

3.3.2 Analytics-aware Network Telemetry

Switches are the source of network traffic data (i.e., packet headers or records), as prior network measurement systems [256, 297, 250, 332, 417, 284, 401] have observed. Offloading network analytics tasks directly to line-rate PFEs on network switches is therefore appealing but comes with previously explained drawbacks (Section 3.2.3). Unlike prior systems, Jetstream does not push any application specific logic down to the switch level. Instead, we leverage programmable data plane technology for offloading functionality common across all applications, such as compressing, distributing, and load balancing telemetry data streams.

Jetstream's data plane frontend builds on *Flow [402], an existing high-performance network telemetry platform that exports digests containing per-packet measurements. We elaborate on how we extend and make *Flow **analytics-aware** by implementing application-specific, runtime-configurable distribution and load balancing of telemetry streams in Section 3.4.

3.3.3 Highly-parallel Streaming Analytics

The streaming analytics engine performs the vast majority of analytical computations touching on every single exported packet in software. This is the core component of Jetstream, supporting custom applications implemented as stream processing programs.

In the stream processing paradigm, an application is a graph (or **pipeline**) that is organized in several stages. Each stage performs one computational task and is implemented using one or many parallel **kernels** (or **operators**) that transform (e.g., map, filter, or reduce) an unbounded stream of tuples [449, 169, 322, 17]. In traditional stream processing, applications scale at the stage-level. Each operator typically runs in a separate thread and maps to a physical processor core. This model is a clean and simple abstraction for data processing applications, but presents two main challenges.

First, it requires load balancing between kernels in software, which introduces bottlenecks described in Section 3.2.4. We overcome this challenge by scaling at the granularity of full pipelines. An application consists of multiple independent pipelines that each handle a distinct subset of flows. Jetstream's data plane component partitions packet records between these pipelines and encapsulates each record in a UDP packet. The UDP destination port encodes the application instance selected to process the packet. At the analytics server, the NIC uses the port number to select the appropriate hardware queue for each packet; each Jetstream pipeline then only ever reads from its assigned queue.

Second, stream processing platforms add communication and data structure overheads. We address this challenge by carefully applying a set of software optimizations that are adapted and tuned for packet-level network analytics workloads. These optimizations have the goal to minimize the amount of costly copy operations, improve data locality within processing pipelines and amortize remaining, inevitable cost by using batching. We elaborate on the unique characteristics of packet analytics workloads and their impact on our design and optimization choices in Section 3.5.

3.3.4 On-demand Metric Aggregation and Analysis in Backend Systems

Finally, the results of stream processing pipelines, which will generally consist of high-level information at significantly lower rates, can be fed into different backend systems, such as security platforms as alerts [373], time series databases for visualization, auditing, offline analysis [106], or network controllers for automated network reconfiguration [255]. In order to mine meaningful and network-wide metrics and analytics results, event streams must eventually be merged and aggregated across analytics pipelines and servers. As explained before, this is costly when done within the stream processor and at rates of millions of records per second but becomes feasible when performed on event streams of hundreds or even thousands of records per second and outside of the critical analytics pipelines.

Consequently, to maintain pipeline independent processing, we push cross-pipeline data aggregation into the backend itself. As a proof of concept, we use a time series database system which is already optimized to aggregate data from many sources. Each metric calculation pipeline streams data directly into a database proxy, which exposes per-instance flow metrics through an interface that the database scrapes. We show that our model fits existing time series database systems well and dive into each phase of the on-demand aggregation and analysis part of our system in Section 3.7.

3.4 Analytics-aware Network Telemetry

The Jetstream data plane interface connects line-rate telemetry systems with the Jetstream analytics processing servers. As we view compression as an important system-level functionality to support, we build our prototype with concepts taken from *Flow [402], which emits **grouped packet vectors** (GPVs). A GPV is simply a variable-length list of packet features grouped by flow for more efficient processing with software. One can think of GPVs as a deduplication-based compression format for packet records. In an evaluation of a wide-area Internet packet trace collected by CAIDA [108], using GPVs results in a 7.7x reduction in bandwidth over packet records

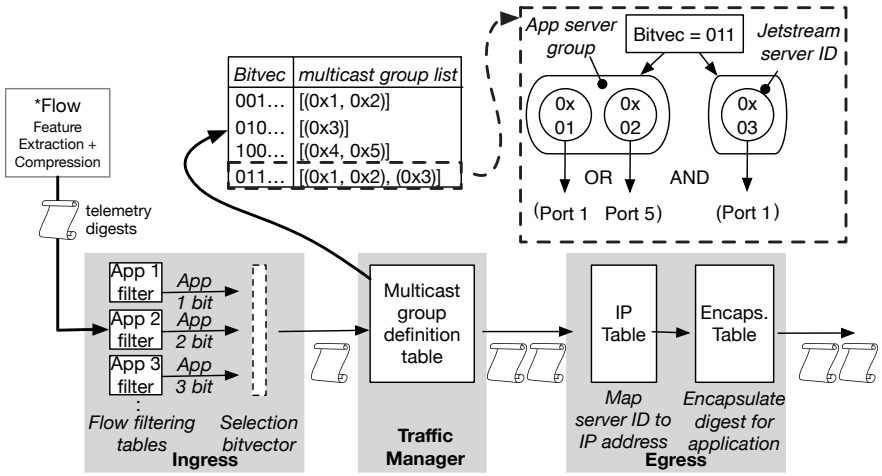


Figure 3.4: Jetstream's data plane frontend for filtering, replication, and load balancing of telemetry digests written in P4

(which already provide significant compression over full packet traces).

Jetstream's data plane component, illustrated in Figure 3.4, extends *Flow to distribute and load balance GPV streams to application pipeline instances, solving the problem of getting the right telemetry streams to the right analytics servers efficiently. This, in turn, eliminates the first bottleneck of general software stream processing for network analytics and allows application pipelines to operate entirely in parallel.

We leverage programmable switches (e.g., in our case the Barefoot Tofino [273]) to support three important functions at line rate: **replicating** streams of telemetry digests to multiple concurrent applications; **filtering** each application's stream to only contain relevant packet flows; and **load balancing** each application's stream across an arbitrary number of stream processing pipeline instances. We implemented these three functions on top of the feature extraction and compression functionality of *Flow.

Usage. The abstraction for the Jetstream data plane interface is simple and application-centric. Each application configures match+action tables used by the Jetstream P4 [177] program to set the IP addresses of its Jetstream processing servers. The switch will load balance telemetry digests destined for the application across these servers, based on a key. The key can be configured per application and is generally the IP 5-tuple or a subset of it. For example, for an application

that only computes statistics per destination IP address, using only the destination IP address as the key means that packets with a particular destination IP address would always end up at the same Jetstream pipeline eliminating the need for later data aggregation. Each application also configures a dedicated filtering table that specifies which flows it needs to monitor. Only telemetry digests matching the filtering table are cloned to the application's servers. The filtering tables can either use exact or ternary matching over the flow key. A new application is added and configured at runtime by populating entries in the respective match+action tables using the RPC mechanisms exposed by the data plane target [87]. This means that adding, scaling or removing a Jetstream application does not require reloading the data plane as it is required in existing systems [332] incurring switch downtimes of tens of seconds [402].

Design. As illustrated by Figure 3.4, the Jetstream data plane interface is implemented as a sequence of match+action tables in the ingress, multicast engine, and egress stages of a programmable switch. The input is a stream of telemetry digests from *Flow or any other data plane telemetry system. During the ingress stage, Jetstream applies a set of parallel match+action tables to determine which set of applications need to process each digest, based on its flow key. Each table holds the filtering policy of one application and sets a single bit in a **flow selection bitvector** packet metadata field, i.e., `bitvec[2] == 1` means that the third application needs a copy of the current digest.

After ingress, the digest and flow selection bitvector proceed to the switch traffic manager. The traffic manager (TM) uses the bitvector as a reference into its multicast configuration table. For modern switches, e.g., the Barefoot Tofino, each entry in this table maps a multicast ID to a set of multicast groups. As Figure 3.4 shows, Jetstream configures this tree structure so that each group represents the servers where a specific Jetstream application runs. The TM selects one member of each group using a hash of the load balancing key, clones the digest to the associated port, and annotates the packet with the ID of the selected member. Each ID is a 16-bit value, which we configure to encode the ID of a specific analytics server. Finally, in the egress pipeline, the switch encapsulates each replica of the digest. To determine the destination IP address, it uses

a table that maps the Jetstream server ID to an IP address.

While we use *Flow as the underlying telemetry system, it is important to note, that Jetstream's data plane frontend is flexible and can be used with any data plane based telemetry source. For example, previous systems, such as Marple [332] and Sonata [250] can be integrated as telemetry sources and subsequently highly benefit from Jetstream's software processing performance and capabilities.

3.5 High-Performance Stream Processing of Network Records

The Jetstream data plane frontend sends telemetry records directly to the individual stream processing pipelines of one or more applications. This allows the pipelines to avoid interaction for distributing network records in software (i.e., the first bottleneck in Section 3.2.4) and enables us to focus entirely on optimizations for the workload. In this section, we explore some of the distinct characteristics of packet analytics workloads and describe how we can leverage them to reduce communication and data structure overheads.

3.5.1 Packet Analytics Workloads

We identify six key differences between network packet analytics workloads and those of general stream processing.

High Record Rates. One of the most striking differences between packet analytics workloads and typical stream processing workloads are higher record rates. For example, Twitter reports that their stream processing cluster handles up to 46 million events per second [424, 425]. For comparison, the aggregate rate of packets leaving their cache network is over 320 million per second; and this only represents approximately 3% of their total network.

Small Records. Although record rates are higher for packet analytics, the sizes of individual records are smaller, which makes the overall bit-rate of the processing manageable. Network analytics applications are predominately interested in statistics (metrics) derived from packet headers and processing metadata, which are only a small portion of each packet. A 40 Byte packet record,

for example, can contain the headers required for most packet analytics tasks. In contrast, records in typical stream processing workloads are much larger.

Event Rate Reduction. Packet analytics applications often aggregate data significantly (e.g., by connection) before applying heavyweight data mining or visualization algorithms. This is not true and applicable for general stream processing workloads, where the backend algorithm may operate on features derived from each record.

Simple, Well-formed Records. Packet analytics records are also simple and well-formed. Each packet record has the same size and contains the same fields that can be accessed in constant time when in memory. Within the fields, the values are also of fixed size and have simple types, e.g., counters or flags. Records are much more complex for general stream processing systems because they represent complex objects, e.g., web pages, free-form text, and are encoded in serialization formats such as JSON and protocol buffers that require more complex parsing.

Network Attached Input. Data for packet analytics comes from one source: the network. Be it a router, switch, or middlebox that exports records, they will ultimately arrive in software via a network interface. In general stream processing workloads, the input source can be anything: a database, a sensor, or another stream processor.

Partionability. There are common ways to partition packet records that are relevant to many different applications, for example, using the flow key (e.g., IP 5-tuple) for load balancing. Further, since the fields of a packet are well defined, the partitioning is straightforward to implement. In general stream processing workloads, partitioning is application specific and can require parsing fields from complex objects.

3.5.2 Jetstream Optimizations for Packet Analytics Workloads

Based on the observations about packet analytic workloads, we identified five important components of stream processing systems where we apply optimizations in Jetstream. We measure the benefit of these optimizations in Section 3.8.1.

Data Input. In general-purpose stream processing systems, data can be read from many

sources, such as a HTTP API, a message queue system (e.g., RabbitMQ or Kafka), or a specialized file system like HDFS. These frameworks can add overhead at many levels, including due to context switches and copy operations. Since packet analytics tasks all have the same source, the network, a stream processing system designed for packet analytics can use kernel bypass and related technologies, such as DPDK [37], PF_RING [338], or netmap [372], to reduce overhead by mapping the packet records directly to buffers in the stream processing system. Jetstream uses netmap [372] to read packet records from individual NIC queues directly into the stream processor without introducing overheads from the operating system networking stack.

Zero-Copy Message Passing. Through our initial experiments, we have identified that for most applications the performance of a single processor within the stream processing graph is I/O-bound. Specifically, frequent read, write, and copy operations into the queues connecting kernels introduce significant performance penalties. Since packet records are small and well-formed, Jetstream can eliminate this overhead by pre-allocating buffers and passing pointers between kernels, to significantly improve performance. In Jetstream, for the output of kernels that do not alter the record data structure (e.g., filter operations), we amortize data copy overheads by using pointers together with C++ move semantics [408] that allow the compiler to avoid deep copies.

Concurrent Queues. Elements in a stream processing pipeline communicate using queues, which can themselves have significant impact on overall application performance. We identified thread-safety and memory layout as primary bottlenecks in queue implementations. For example, basic concurrent queue implementations use expensive locks to ensure thread safety and use linked lists as their underlying data structure. Linked lists allow automatic resizing of the buffer but are expensive due to poor cache locality and frequent pointer dereferencing. Jetstream's design, in which stream distribution and load balancing is offloaded to the data plane, means that most queues connect a single producer and consumer. Based on this insight and leveraging several techniques used in concurrent data structures [435], we implemented an efficient, lock-free ring buffer. As records are small, we use a flat memory layout to avoid overheads of frequent pointer dereferencing. This means that the entire ring buffer is allocated as a single fixed size array and the array cells hold

the actual data tuples as opposed to pointers to the data. The size of the ring buffer is also locked to powers of two allowing for cheaper bit shift operations instead of modulo operations to calculate offsets [441]. Finally, we use atomic types for head and tail indices to enable thread-safety [408].

Hash Tables. Often, network analytics applications need to map packet records to prior state. This requires a key-value store, which can easily be a bottleneck when processing high-rate packet streams. As a solution, Jetstream's library provides an optimized data structure that exploits the fact that packet records are small, well-formed, and have fixed width fields. The reduce operator and a flow table component that are commonly used by network analytics applications and are part of the Jetstream library use a hash table with a flat memory layout and open addressing with linear probing to reduce the overhead of pointer dereferencing and increase cache hit rates. Additionally, to minimize the cost of key comparisons during lookups, Jetstream's hash table encodes keys using 128-bit integers so that they can be compared using a single *Streaming SIMD (SSE)* vector instruction [4, 272].

Batching. Finally, the small size of individual network records makes batching appealing and improves performance in multiple ways. Batching access to queues amortizes the cost of individual queue and dequeue operations. Batching packet records by flow, as done by Jetstream's *Flow-based telemetry data plane, amortizes the cost of hash table operations necessary for a wide range of aggregation tasks that use the flow key or a subset of it as the aggregation key.

3.6 Programmability and Applications

Jetstream analytics applications are written in C++, a popular, general-purpose language enabling easy prototyping, testing, and deployment. Applications leverage the Jetstream library of optimized stream processing primitives. This library not only includes the stream processing core and runtime, but also a variety of pre-built processors that can be used to rapidly build network monitoring and analytics applications. Additionally, application developers can define custom processors.

Processor	Parameters	Output	Description
filter¡*In*¿	function¡bool(*In*&)¿ *p*	*In*	Filter out records of type *In* that do not satisfy boolean predicate *p*.
gpv_receiver	string iface_name	gpv	Receive GPVs from network interface *iface_name*.
join¡*In1,In2,Out*¿	function¡bool(*In1*&, *In2*&) *c*, function¡*Out*(*In1*&, *In2*&) *m*	*Out*	Joins two streams on matching condition *c* emitting tuples of new type *Out* assembled by *m*.
map¡*In,Out*¿	function¡*Out*(*In*&)¿ *f*	*Out*	Apply function *f* to inputs of type *In*, emitting tuples of type *Out*.
print¡*In*¿	ostream& os	void	Print a summary of type *In* to a C++ output stream os.
reduce¡*K,V*¿	function¡*V*(*V*&,*V*&)¿ *r*	pair¡*K,V*¿	Reduce values of type *V* grouped by keys of type *K* using reducing function *r* (e.g., std::plus to sum values by key).

Table 3.1: Processors in the Jetstream standard library (namespace prefixes *js* and *std* are omitted)

3.6.1 Input/Output and Record Format

As Jetstream's telemetry frontend extends a prior telemetry system, *Flow, we leverage *Flow's record model, grouped packet vectors. Unlike traditional flow records, GPVs still contain individual packet data (such as individual timestamps, byte counts, or TCP flags) through feature vectors. We leverage GPVs that include individual microsecond timestamps, byte counts, hardware queuing delays, queue ids, queue depths, IP ids, and TCP sequence numbers. Further information on the GPV format and GPV generation in both software and hardware can be found in [402].

The primary packet input mechanism in our system leverages netmap [372], a kernel-bypass mechanism allowing the mapping of NIC buffers directly into the stream processor's (user space) memory. Using this, we are able to inject packet records at high rates into the Jetstream analytics system without allowing costly and frequent system calls to become a bottleneck in the processing pipeline. While kernel-bypass NIC access is the primary packet interface in our system, we also implemented the ability to read GPVs from memory, from files, from standard sockets, or to receive raw packet records using PCAP or the TaZmen sniffer protocol.

Function	Description
a.add_stage¡*P*¿(*args...*)	Add stage with processors of type *P* to application *a*; initialize the processor with arguments *args...*.
a.connect¡*T*¿(*s1*,*s2*)	Connect stage *s1* emitting type *T* with stage *s2* in the processing graph of application *a*.
a()	Run application *a*

Table 3.2: API for composing and running applications

3.6.2 Programming Model

Jetstream applications are written as stream processing pipelines. The simplest way for a developer to write an application is by composing Jetstream's builtin stream processors, for example those listed in Table 3.1. Table 3.2 shows Jetstream's API for interconnecting these processors and launching pipelines. A simple application counting the number of packets per source IP address can be defined like this:

```
js::app a;
auto rx = a.add_stage<js::gpv_receiver>("enp2s0f0");
auto map
  = a.add_stage<js::map<gpv,pair<js::ipv4_addr,unsigned>>
    ([](gpv x){return std::make_pair(x.ipsrc,x.pktcount);});
auto reduce
  = a.add_stage<js::reduce<js::ipv4_addr,unsigned>>(plus());
a.connect<gpv>(rx,map);
a.connect<pair<js::ipv4_addr,unsigned>>(map,reduce);
a();
```

Here, `js::app a;` declares and instantiates a pipeline (or application). Calls to `a.add_stage()` and `a.connect()` define the pipeline, and its execution begins on the last line when we call the function operator (`a()`). Using this API, each application defines the processing steps it requires.

Jetstream includes a standard library (short `js`) of common processors that can be chained to build full network analytics applications. The library includes common data flow operations listed in Table 3.1. Additionally, specialized domain-specific operators exist to, for example, reduce

by flow key (i.e., a flow table). All processors in the Jetstream library leverage different software optimizations outlined in Section 3.5.2.

3.6.3 Custom Processors

If an analytics task requires processing logic, data types, or interfaces that are not covered by Jetstream's library, developers can implement custom processors that automatically take advantage of Jetstream's scaling and load balancing.

To write a custom processor, a developer first creates a subclass of `js::proc`. Next, the developer specifies input and output ports and types in the subclass's constructor. These ports are used to send or receive records to or from other processors, respectively. Finally, the developer implements processing logic in the `operator()` method. For example, a basic version of the *print* processor from Table 3.1 can be implemented like this:

```
class print : public js::proc {
public:
   print() { add_in_port<gpv>(0); }
   bool operator()() {
     gpv gpv; js::signal sig;
     in_port<gpv>(0)->dequeue_wait(gpv, sig);
     _os << gpv << std::endl;
     return sig == sig::proceed; }
private: std::ostream& _os; };
```

Custom processors allow developers to implement arbitrary applications that operate on packet records or GPV inputs. They are written as standard C++ code and can use custom algorithms and data structures, leverage third party libraries, or call external services.

3.7 On-Demand Aggregation in Backend Systems

Processing pipelines in Jetstream are optimized to efficiently extract higher-level information from the input data packet stream. We refer to this higher-level information as **metrics**. In our prototype implementation such metric tuples consist of a numeric value, a timestamp, and

a set of key-value metadata pairs. For example, to detect elephant flows, a heavy hitter detector implemented on top of Jetstream would periodically export the number of packets or bytes together with flow information (i.e., the IP 5-tuple) for the most active flows [279].

3.7.1 Integrating with Backend Systems

In Jetstream, the final aggregation of computed metrics is offloaded to configurable backend systems. This is possible as long as the analytics application already significantly (i.e., by several orders of magnitude) reduces the event rate. Intuitively, this is the common case for analytics applications because useful metrics aggregate data (e.g., in small time intervals), or report on anomalies that are by definition infrequent.

The backend can then be used to automatically or interactively query, analyze, or visualize metric data computed by Jetstream. Jetstream integrates with backend systems through an API that can be used by applications to export metrics from pipelines. A local metric collection proxy consumes app metrics and exposes an interface that can subsequently be polled by the backend system. The export API used within Jetstream applications is universal while the API exposed by the collection proxy is specific to the respective backend system. We imagine possible backend systems to be time series databases (e.g., Prometheus [72]), visualization systems (e.g., Grafana [52]), monitoring platforms (e.g., Nagios [80]), another stream processor, or a network control platform (e.g., ONOS [82]) to enable a network control loop.

Exporting Metrics. The metrics export API currently supports two types of metrics inspired by the Prometheus time series database: a *counter* and a *gauge*. A counter metric represents a cumulative and monotonically increasing value while a gauge can be set to a specific value, increased, or decreased in value. Each metric is associated with a name, a timestamp, and a set of meta data. Other metric types, such as snapshots of full metric distributions or vectors are imaginable. For example, upon detection, reporting a heavy hitter using a counter from within a Jetstream pipeline looks like this:

```
js::metrics.update_counter("heavy_hitters", hh.pkt_count,
```

```
{{"ip_src", hh.ip_src}, ... });
```

Collection Proxy. Internally, the metrics export API adds a timestamp, serializes the metric object using Protocol Buffers [92], and sends a RPC message using gRPC [53] to the collection proxy. The collection proxy sits between a Jetstream application and the backend system, converting data into the appropriate format. In order to prevent data aggregation in-line resulting in cross-core communication, a collection proxy is instantiated for each instance of a Jetstream application and subscribes to an instance's metric stream. We built a collection proxy prototype for the Prometheus time series database [72]. For this integration, the proxy exposes a HTTP API that a Prometheus instance periodically scrapes. Finally, Prometheus stores scraped metrics in its data store for continuous aggregation across Jetstream instances.

3.7.2 Querying Metrics

Prometheus supplies a query language and API, which allows a user to retrieve network traffic metrics from the database. Additionally, Prometheus allows configuring alerts and integrates with Grafana [52], a framework to easily visualize query results to, for example, build dashboards. All our example applications integrate with the metrics export system and can be queried from Prometheus. We now show example queries for three of those applications to illustrate how a user can interact with and extract relevant metrics from Jetstream.

For the traffic accounting application, Prometheus maintains individual counters for each component of a packet's IP 5-tuple. For example, a user can use the Prometheus `rate()` function to calculate the average number of bytes per second sourcing from port 443 over the last minute using this query:

```
rate(total_bytes{tp_src="443"})[1m]}
```

The heavy hitter application, which looks for flows that cause more than a configurable percentage of the total bytes in the network, exports heavy hitter candidates with the metric name

`heavy_hitters`. In order to identify the top 5 frequent flows from the candidates stored in the database, we can issue a query as follows:

```
topk(5, heavy_hitters)
```

The TCP analysis application looks for out of order packets in a TCP flow. Flows with at least one out of order packet are exported to the database with the metric name `tcp_seq` and the metric value counting the number of out-of-order packets in the flow. To find which flows originating from the 192.168.0.0/16 subnet have more than 10 out-of-order packets, the user can issue the following query:

```
tcp_seq{ip_src=~"192.168.+.+"} > 10
```

3.8 Evaluation

We evaluate the performance and efficacy of our prototype implementation through three different lenses. First, we measure Jetstream's overall system throughput and scalability from both an end-to-end standpoint as well as from an individual application throughput standpoint. We then look at how Jetstream's telemetry-aware data plane component compares with Sonata [250] in terms of PFE resource consumption and accuracy. Finally, we evaluate the performance of Jetstream's stream processor against both Spark [449] and dShark [442].

We used the Cloudlab network experimentation platform [221] for all of our benchmarks. Our experiment setup consisted of six servers with 2 $\times$ 10-core Intel Xeon E5-2660 v3 CPUs clocked at 2.6 Ghz. Each node had 160GB of ECC DDR4 memory. The nodes were connected over a 10Gbps network with two Intel X520 Ethernet adapters per server for ingestion of telemetry data. We used packet traces from a core Internet link collected by CAIDA in 2015 [108] for all experiments.

3.8.1 Macro Benchmarks

First, we benchmark Jetstream's performance and scalability at a macro-level using the applications described in Table 3.3. In this experiment, we created a scenario where three switches

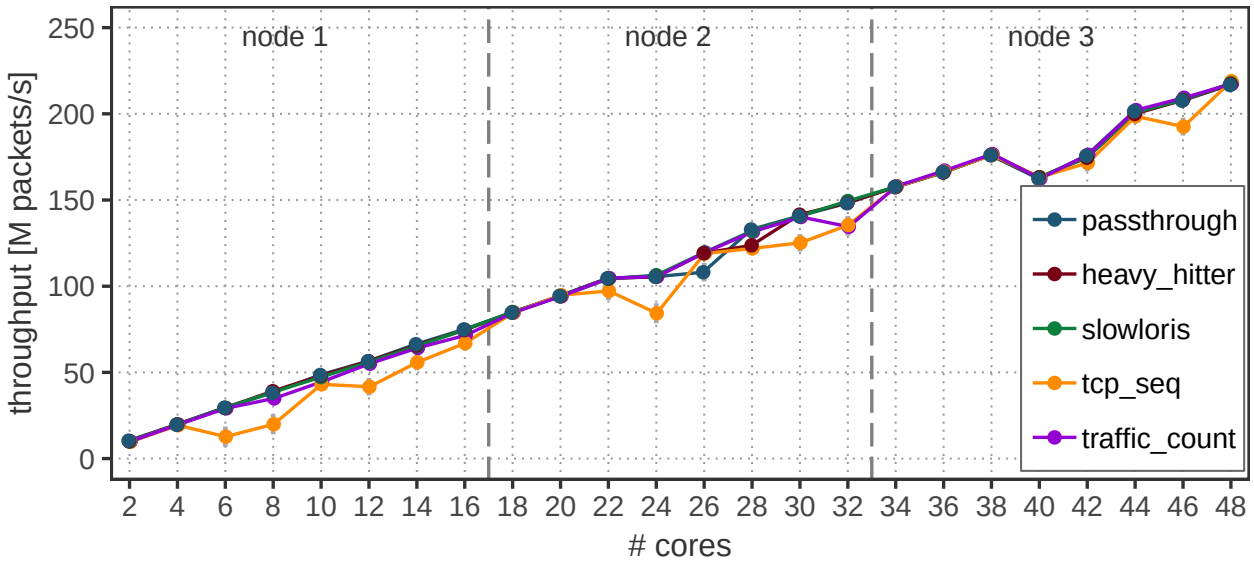


Figure 3.5: Scalability of Jetstream applications across servers

stream GPVs across the network to three Jetstream analytics servers running application pipelines. Our programmable switch (Tofino) is currently not physically co-located with sufficient server resources. We therefore model the switches by running a software implementation of Jetstream's data plane component on three separate servers in the Cloudlab network, driven by real-world packet traces from CAIDA. Each pipeline uses two cores scaling to a total of eight pipelines per server, or 24 pipelines using 48 cores across three servers. Each 10GbE network interface serves up to four Jetstream pipelines.

Figure 3.5 shows the performance and scalability of Jetstream. We ran 24 rounds of this experiment where we added an additional analytics pipeline (2 cores) with each round, eventually using all 3×16 cores of our servers. Our system scales linearly with core count across machines and can process over 200 million packet records per second leveraging only three commodity servers. This demonstrates the effectiveness of key design choices in Jetstream.

The bottleneck in this set of benchmarks was the 10Gbps network interface card we used. With the assumption that telemetry packets are roughly 200 bytes on average (since a packet is a GPV record), the max rate of a 10Gbps network interface would be about 6M GPVs/sec. We had each of the two NICs feed 4 of the pipelines (8 of the cores), which led to roughly 1.5M GPVs/sec

Application	Mean	Description
Passthrough	31.8	Simple GPV passthtough (no ops.)
Traffic Count	14.0	Count total GPVs, pkts, bytes
Heavy Hitter	16.2	Find IPs sending ¿θ% of total packets in network
TCP Seq.	15.0	Find TCP flows with out of order packets
Slowloris	14.7	Find IPs w/ many low traffic volume TCP connections
SLB Profiler	31.6	Software load balancer
SSH Brute Force	10.8	Identify SSH Brute Force attacks
SYN Flood	21.1	Identify SYN Flood DoS attacks

Table 3.3: Jetstream's per-application throughput [M pkts/s]. Two cores per application.

per pipeline. With an average of 8 packets per GPV in the trace that we used, this translates to roughly 12M packet records per second per pipeline that we can theoretically feed per pipeline, or a maximum theoretical rate of 96 M packet records per second per server with two 10Gbps NICs. In practice this rate is likely lower due to a variety of factors. We saw roughly 75M packet records per second in practice of just I/O performance. As we will see next, many of our applications scale beyond this number and would therefore benefit from higher throughput NICs.

To show the performance of the individual Jetstream applications without the NIC input bottleneck in our setup, we also stream network traffic from memory through Jetstream. Again, in this experiment, each application is assigned two cores as each application has one thread dedicated to consuming records while the other thread runs the application. Table 3.3 shows Jetstream's application performance numbers. We can see that Jetstream achieves a maximum throughput in excess of 31 million packets per second per pipeline while also attaining strong performance for complex, stateful applications such as SSH Brute Force detection. As a result, Jetstream pipelines process data between 1.5 to 3 times faster than the 10Gbit/s telemetry input over the network. In practice, a 40 Gbit/s NIC would be able to fully utilize the analytics pipelines.

Jetstream's high processing rates are a result of applying the different software optimization strategies outlined in Section 3.5.2. Using GPVs provided a speedup of 5.4 over single packet records. Our optimized concurrent queue implementation was faster by a factor of 3.0 over the C++ standard template library queue (secured with locks). Our hash table implementation using

# Apps	Stages	Tables	VLIWs	Metadata	SRAM	TCAM
1	2	5	3	888b	128KB	3.84KB
4	2	8	5	912b	128KB	15.36KB
8	2	12	7	944b	128KB	30.72KB
12	3	16	9	976b	128KB	46.08KB
16	3	20	11	1008b	240KB	61.44KB

Table 3.4: Jetstream network interface resource usage on the Barefoot Tofino. Stateful ALU usage is 0 for all applications.

a flat layout and linear probing provided a speedup of 1.8 over the STL standard unordered map. Finally, using netmap instead of standard sockets provided a throughput increase of a factor of 2.8. To obtain these numbers each optimization was isolated from all others.

3.8.2 Comparison with Hardware Analytics

We next evaluate Jetstream's data plane component, a line-rate data plane program written in P4 that filters, replicates, and load balances telemetry digests across analytics servers. We ran this program on a Barefoot Tofino PFE configured with ternary application filtering tables sized for 128 entries each. Table 3.4 lists the major resource requirements of the Jetstream data plane interface. Overall, the component is lightweight: It requires only 3 stages and 20 tables to filter for 16 different applications because of its parallel design. The most-utilized resource is TCAM. Each application's table uses approximately 1% of the Tofino's TCAM. If wildcard and priority-based filtering is not required for all applications, some or all of the tables can be replaced with exact match tables in SRAM rather than TCAM.

We now compare Jetstream's PFE resource consumption with that of Sonata [250], a state-of-the-art network telemetry and analytics platform that leverages switch hardware to accelerate network analytics. Sonata's primary goal is to reduce the load on the software stream processor by iteratively refining network queries and pushing them into hardware.

While Sonata is able to reduce the event rate at the stream processor, the system makes tradeoffs to realize this performance. First, Sonata's iterative refinement reduces the required state maintained by the switch to execute a query. However, refinement comes at the cost of an increasing

Query	Stages	Tables	VLIWs	Metadata	sALUs	SRAM
Heavy Hit.	5	13	7	912b	1	112KB
New Conn.	6	16	8	1032b	1	128KB
S. Spreader	8	19	9	840b	2	192KB
Port Scan	8	20	10	1072b	2	208KB
SSH Brute	9	26	11	984b	2	224KB
SYN Flood	11	25	17	1312b	2	288KB
Cmpl. Flows	11	26	17	1312b	2	304KB
Slowloris	11	27	17	1316b	3	336KB
With one level of refinement (Sonata)						
Heavy Hit.	7	22	11	1152b	2	224KB
New Conn.	9	28	13	1184b	2	256KB
Others	***Compilation failed, insufficient resources***					

Table 3.5: Resource usage for hardware analytics queries on the Barefoot Tofino. SRAM requirement assumes <65K concurrent keys (e.g., one 10 Gb/s Internet link [108]).

number of match+action tables to perform the same query. Table 3.5 illustrates this point, as many queries that run with multiple levels of refinement fail to compile to the switch. If we compare Sonata (Table 3.5) and Jetstream's (Table 3.4) resource usage, we can see that Jetstream only requires about as many resources (stages, tables, etc.) as a single Sonata query in hardware, even to support expensive load balancing and filtering for many concurrent applications.

The second of Sonata's tradeoffs also stems from query refinement and results in a reduction in accuracy. Each iteration of refinement that reduces load on the stream processor, requires another time window to pass by before packets are forwarded to the stream processor. As a result, in order to get the largest reduction in event rate at the stream processor, applications must wait multiple time windows before being able to process potentially time-critical data. Waiting one or more time windows negatively impacts accuracy for many applications as issues lasting fewer than one or more time windows (e.g., frequent micro-bursts [389]) will not be detected. Jetstream has no such accuracy limitation since processing is done in software. Detection performance is predictable and attacks will not slide through the cracks.

Finally note that, while Jetstream provides a telemetry replacement for Sonata at a lower PFE resource cost, Sonata (or other telemetry systems) and Jetstream can technically be used in conjunction. This may be beneficial in certain cases, e.g., when a simple, static query fits entirely in the PFE. Doing so, however, sacrifices flexibility. For example, it makes runtime reconfiguration more challenging (see Section 3.2.3).

3.8.3 Comparison with Pure Software Analytics

In this chapter, we argue that software provides the programmability and flexibility needed to support a wide range of network analytics applications. Existing software analytics platforms, however, do not provide sufficient performance for cloud-scale network analytics workloads. To support our argument, we now compare the performance of our system against both Spark (used by Sonata [250]) and dShark [442]. For each test, we used the same experimental setup as described in Section 5.6.

General-purpose Analytics (unmodified Spark)

To illustrate the impact that just the architectural changes have, we compare against Spark [449], a general-purpose stream processing system. We ran the Traffic Accounting application, which counts the number of packets and bytes per each component of the IP 5-tuple. We streamed GPVs as input data over the network to both Spark and Jetstream. With two CPU cores, Spark sustains 1.4 million packet records per second, whereas Jetstream runs at 9.9 million packet records per second. Most importantly, we found that for this workload Spark (unlike Jetstream) does not scale with core count (or number of threads). Spark's inability to scale in this scenario is due to the high-volume input streams in network telemetry that Spark distributes across worker threads in software. This imposes very high utilization in the distribution/load balancing thread and subsequently creates a bottleneck. In Jetstream, this critical task is offloaded to programmable line rate switches. We gave more intuition on this in Section 3.2.4. Other Spark users have also found Spark to scale poorly for comparable workloads [102], confirming our tests.

General-purpose Analytics (Spark with kernel bypass)

Of course, a question arises if Jetstream's benefit just comes from its use of kernel bypass technology. As it is non-trivial to modify Spark to include streamlined network I/O capabilities, we use streaming from memory within the application as a way to remove the I/O component from the evaluation. That is, we read an entire trace into memory and replay it directly within the application. With 2 cores, Spark runs at 2.0 million packet records per second, whereas Jetstream

runs at 14.0 million packet records per second, further illustrating Spark's architectural bottleneck.

Network Analytics Software (dShark)

To understand Jetstream's true software processing performance in the face of the NIC bottlenecks we experienced, we compare against dShark [442], a recently introduced software-based, packet-level, network analytics platform. A key innovation of dShark was the ability to analyze traffic in the face of network packet header transformations. One such application which requires this functionality is the software load-balancer (SLB) profiler in dShark. We re-created the SLB profiler application in Jetstream and validated its correctness in a live test. Our results illustrate Jetstream's comparable flexibility to dShark. We acknowledge, however, that because Jetstream relies on GPVs, which are fixed-format records, we can only support a fixed depth of header nesting, whereas dShark can support any depth. We believe this limitation is not impactful for this discussion, as in practice, it would be highly irregular to see a depth of nesting beyond some known amount. Since dShark is not open source, we reference the performance results in their publication. While not a perfect comparison, our results are still illustrative with Jetstream running on similar hardware. In the dShark experiments, packet records are streamed from memory directly into the analytics application. On a 16-core server, dShark runs at 10.6 million packets per second (Mpps) with 6 parsers and 9 groupers (or 0.625 Mpps per core), whereas Jetstream runs at 31.6 Mpps (or 15.9 Mpps per core), a 25.44x speedup. Here, we note that performance scales linearly with number of servers in both cases.

Resource Cost Analysis

To put the performance speedups into context, consider the resources needed to support analytics in a modern datacenter. Here, we look at reported traffic in a cluster at Facebook [376] where an analytics system needs to sustain at least 961 million packets per second in order to meet the web server cluster's peak packet rates. Assuming 16-core servers, we would need ˜96 servers for each analytics application to run on dShark, ˜480 servers for systems using Spark, and a mere 4 servers for systems using Jetstream. These numbers also assume that dShark and Spark integrate optimized packet input through, for example, kernel-bypass technology, as Jetstream does.

3.9 Conclusion

This chapter introduced Jetstream, a high-performance platform for network analytics that makes no compromises on performance or generality — records of every packet can efficiently be processed in software. Jetstream enables fine-grained control over a developer's network monitoring application. Developers can now optimize their applications' network security, network performance, and network efficiency. The core insight of Jetstream is to utilize programmable networking hardware to improve the performance of software analytics platforms, rather than offloading analytics applications themselves.

The resulting prototype of Jetstream can analyze between 86.4 and 254.4 million packets per second on a 16-core commodity server. Benchmarks show that Jetstream's approach to telemetry data distribution and load balancing in the data plane enables linear scaling with addition of servers and only requires moderate switch resources. Compared with a high-performance network analytics software system (dShark), Jetstream supports over 25.4x higher processing rates. To process a published data center workload, this would require 96 servers in dShark, but only 4 in Jetstream — making fully flexible software-based network analytics practical.

Chapter 4

Towards the Advancement of Network Intrusion Detection Systems

The creation of Jetstream [324] (Chapter 3) allows us to build scalable, efficient, and high performance network applications that utilize every packet in a flow. One particularly useful application of network monitoring is towards intrusion detection. Here we explore the efficacy of new network intrusion detection systems utilizing per-packet network features to identify malicious internet traffic.

We first utilize PFEs and Jetstream's processing kernels (Section 3.3.3) to build our own network intrusion detection system to identify ransomware via its network traffic signature (Section 4.1). The growth of malware poses a major threat to internet users, governments, and businesses around the world. One of the major types of malware, ransomware, encrypts a user's sensitive information and only returns the original files to the user after a ransom is paid. As malware developers shift the delivery of their product from HTTP to HTTPS to protect themselves from payload inspection, we can no longer rely on deep packet inspection to extract features for malware identification. We utilize PFEs to collect per-packet, network monitoring data at high rates. We use this data to monitor the network traffic between an infected computer and the command and control (C&C) server. We extract high-level flow features from this traffic and use this data for ransomware classification. We write a stream processor and use a random forest, binary classifier to utilizes these rich flow records in fingerprinting malicious, network activity without the requirement of deep packet inspection. Our classification model achieves a detection rate in excess of 0.86, while maintaining a false negative rate under 0.11. Our results suggest that a flow-based fingerprinting

method is feasible and accurate enough to catch ransomware before encryption.[1]

We then utilize these per-packet and per-flow network features to evaluate the efficacy of neural network-based network intrusion detection systems (NIDS). Recently, deep neural networks have been used to identify anomalies in network traffic [144, 446, 440, 310, 326, 457, 451, 191]. However, it has been shown that neural networks are vulnerable to adversarial example attacks in other domains [174, 415]. Adversarial examples are small perturbations of the input that can bypass or purposely alter a neural network's classification. For example, in the case of images, a malicious actor might create an adversarial example by changing a few pixels (imperceptible to the human eye) such that the classifier misclassifies a specific person as a different person or hides that person all together (Section 4.6). Previously proposed anomaly-based NIDSs have not been evaluated in such adversarial settings and the feasibility of crafting adversarial examples from network packets and flows have not been explored. In the latter half of this chapter, we show how to evaluate an anomaly-based NIDS trained on network traffic in the face of adversarial inputs. We show how to craft adversarial inputs in the highly constrained network domain, and we evaluate three recently proposed NIDSs in an adversarial setting.[2]

4.1 Machine Learning-based Detection of Ransomware Using SDN

In recent years, the prevalence of malware, has increased dramatically. In fact, ransomware has grown into one of the most prominent strains of cybercrime. In 2017, we saw more cases of ransomware than we have ever seen before due to its ability to autonomously propagate across the network [227].

Clearly, ransomware mitigation techniques need to be designed in order to prevent successful attacks of malware. Luckily, there has been some work in the detection and mitigation of malware [184, 185, 248]. However, these studies focus on ransomware identification delivered through HTTP. Unfortunately, malware delivery is shifting heavily to HTTPS as 37% of all malware now

[1] Work published at NDSS Poster Session 2018 [209] and SDN-NFV Security 2018 [208]
[2] Work published at AIsec 2018 [258] and Big-DAMA 2019 [208]

utilizes HTTPS as of June, 2017 [307]. We need a longer term approach that utilizes network features only available in TLS traffic. Furthermore, the work in [184] sacrifices the wellbeing of one computer in order to identify malicious servers sending and controlling malware. In this paper, we leverage advances in SDN to address the ransomware problem. Specifically, we utilize the emergence of PFEs (e.g. P4 switches), write a stream processor, and implement machine learning to identify and intercept ransomware before it enters a network.

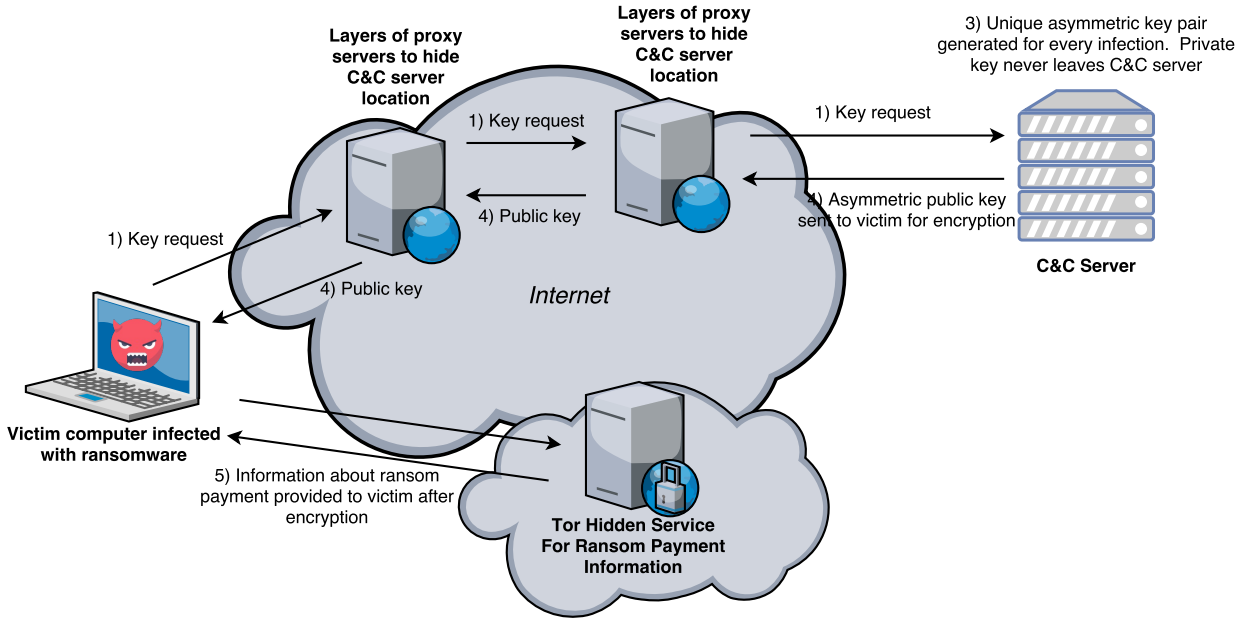


Figure 4.1: Operation of typical ransomware encryption key retrieval process [185].

Ransomware is a software virus that holds a victim's files at ransom. Access to the files is not returned until a ransom is paid. There are two main types of ransomware in circulation today, crypto and locker-based ransomware. Crypto ransomware encrypts the files on a victim's computer and will only provide the decryption key for the files if a ransom is paid. On the other hand, locker ransomware leaves the victim's computer files intact but locks the user out of his or her computer, only returning access once a ransom is paid. Unfortunately, detecting various types of ransomware is an arduous task. Developing a long term solution to ransomware detection has proven difficult since ransomware developers are constantly updating their product to circumvent new detection

techniques. Furthermore, many flavors of ransomware are delivered via botnets [248], and as the IoT sector grows rapidly, the number of avenues for infection are increasing dramatically. We have also seen the emergence of Ransomware as a Service (RaaS), where a novice cybercriminal can pay a service and easily customize his or her own ransomware and have it spread to many computers around the world [423]. Despite the growing number of ransomware cases, the underlying method for how the two methods operate are similar. They both require communication with a C&C server in order to carry out an attack. This communication between the infected computer and the C&C server is what we exploit in our classifier.

Figure 4.1 shows the communication between the infected computer and the C&C server. In order to encrypt the victim's files, the victim requests an encryption key from the C&C server through multiple layers of proxies. The C&C server generates a new asymmetric key pair, keeps the private key, and returns the public key to the victim to encrypt its files. After encryption, a Tor hidden service communicates a method for paying the ransom. By analyzing the traffic flowing between the victim's computer and the proxies residing in the greater Internet, we're able to develop a classification model that identifies the encryption key retrieval process.

Previous work has shown that even if the victim has received the initial infection through a phishing email, for example, if the C&C server cannot deliver the encryption key, the malware cannot carry out the attack [185]. As a result, we look at the network traffic between the victim's computer and the C&C server in hopes that we can identify malicious communication, and prevent the delivery of the encryption key.

In order to accurately monitor all traffic going into and out of the potential victim, we leverage the recent emergence of programmable forwarding engines (PFEs). PFEs utilize switch hardware and dynamic memory caches to achieve high packet processing speeds while simultaneously providing rich flow records. These PFE-generated flow records, provide per-packet information and allow us to extract flow features for ransomware classification at line rate in an accurate and scalable manner.

4.2 Related Work

Two areas of related work help us in designing our ransomware detection application. Ransomware detection has been a large area of study in recent years; however, many of these solutions fall short as ransomware developers adjust their malware delivery methods. We also look at the emergence of PFEs, the programmable hardware we leverage for rapid per-packet, flow processing.

4.2.1 Ransomware Detection

One method of ransomware detection used machine learning to identify and classify various types of ransomware during the ransomware installation phase on target hosts. The authors mainly relied on Windows API calls, file system operations, registry operations, etc. to classify malware. Their ransomware classifier, EldeRAN, was compared to various other machine learning algorithms such as SVM and Näive-Bayes and produced a much higher true positive rate and a lower false positive rate [385]. However, EldeRAN requires the infection of a system in order to learn ransomware behavior.

Another group of researchers used an SDN approach to ransomware identification by utilizing deep packet inspection to track the packet lengths of HTTP POST messages [184]. Once ransomware was identified, the command and control server IP addresses were identified and blocked. However, this technique results in a relatively high false positive rate (up to 4.95%), leaving their method open to a base rate fallacy issue and falsely blocking valid servers.

In fact, most malware and ransomware detection methods that look at traffic traces, like the one above, are payload-based [185, 184, 439]. These network-based approaches to ransomware detection all share the same, previously described problem of relying on DPI, and therefore, are useless for fingerprinting on encrypted traffic.

4.2.2 Recent Hardware Trends and PFEs

In recent years, we have seen the development of a few high rate stream processing systems, which utilize switch hardware to generate network information-rich flows [297, 332, 401, 178]. PFEs allow commodity networking equipment to support the scalable generation of rich flow records. The recent trend of PFEs and the accompanying efforts to make programming them more accessible has enhanced the use and development of PFEs [177].

PFEs allow us to process network data at high rates of speed, while still extracting vital, per-packet flow information. The growth of PFEs and rich flow generation systems, provide us with the data and speed necessary for network, flow-based ransomware classification.

4.3 System Architecture

Our system's architecture is broken into two main parts, stream processing and classification. The stream processor reads from a PCAP, runs and manages a custom flow table, and extracts flow features for our classifier. The classifier takes in the extracted features and trains a model to identify ransomware.

4.3.1 Stream Processing

In order to process rich flow records, we utilize RaftLib's stream processing library to build high-performance, parallel, analytics applications [169]. Each kernel we wrote using RaftLib runs a step in the flow processing chain. We link multiple of our kernels to group incoming packets into their respective flow records based on each packet's 5-tuple. The 5-tuple, which consists of the packet's protocol, source IP, source port, destination IP, and destination port, serves as the flow record's key. The kernel-based approach allows us to utilize RaftLib's parallelization feature. Since we read in network traffic from PCAP files, we use a custom flow table and implement it as a kernel running in parallel with the other kernels. We simulate the generation of rich flow records and use the RaftLib framework to write a parallelized, stream processor for flow feature extraction at line

rate. These extracted features are then used for ransomware classification.

4.3.2 Classification

We implement a random forest classifier in Python due to the random forest's low computational training cost and its use of bagging to reduce variance and overfitting. A random forest classifier is an ensemble algorithm, which utilizes a collection of decision trees to vote and predict the class of the input data. Each decision tree is created from a random subset of the feature set. Each decision tree is generated using the gini impurity metric, which measures the probability of mislabeling a randomly chosen element from the training set if the element was labeled based solely on the distribution of the binary labels in the set [359].

Three of the main tuning metrics for a random forest classifier include the number of decision trees in the forest, the depth of each decision tree, and the maximum number of features that can be included in each decision tree. The number of trees in the forest dictate the performance and variance of the classifier. A larger number of trees results in higher classification accuracy and lower variance but increases the computational cost of the classifier. The depth of each tree has a similar cost-benefit situation. As the depth of each tree increases, the induced bias in the classifier decreases; however, the added depth comes with a computational penalty.

The last main metric we used for tuning our random forest classifier is the maximum number of features that can be included in each decision tree. The maximum number of features is used to determine the best split when creating a decision tree. Once again, increasing the number of features increases performance but comes at a computational cost. In the next section, we discuss our implemented application starting with our stream processor and finishing with the ransomware classifier.

4.4 Implementation

4.4.1 Flow Records and Processing Kernels

We wrote five kernels on top of the RaftLib framework for processing network data and creating compact and rich flow records. Figure 4.2 shows the structure of our flow record. The 5-tuple serves as a key for each flow, which links to the number of packets and bytes in the flow along with a reference to specific packet features. The packet features include the packet timestamp and the number of bytes in the packet. Each flow packet also contains a link to the packet's IP flags and time to live (TTL). We utilize the data in these flow records to extract features for our ransomware classifier.

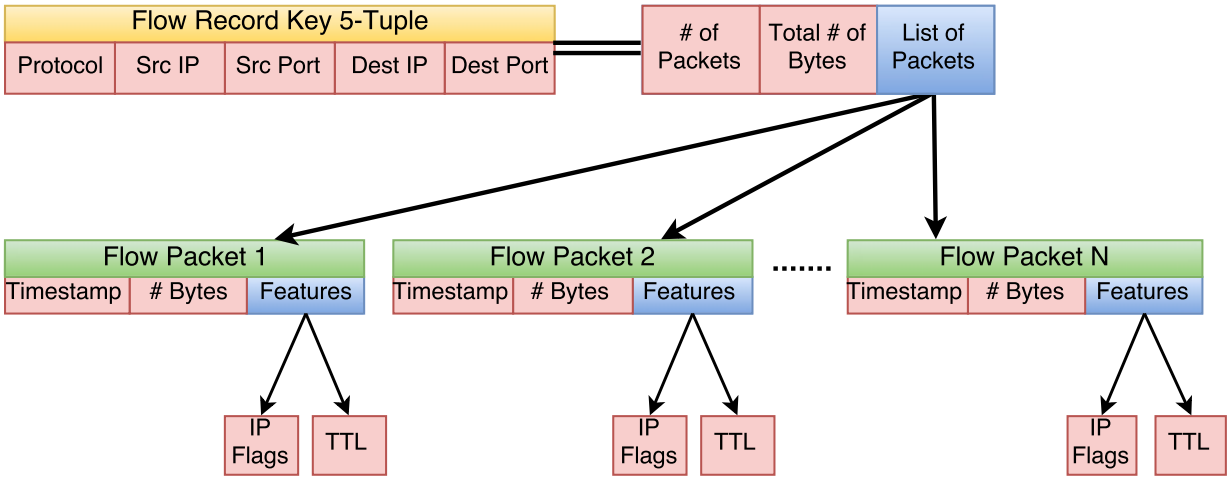


Figure 4.2: Compact and per packet flow records created in a hierarchical manner. The 5-tuple serves as the key for matching packets in the same flow.

Figure 4.3 shows the kernels we wrote for flow generation and feature extraction. Normally, the per packet, flow records seen in Figure 4.2 would be generated in PFE hardware, but since we are reading from a PCAP, we wrote three kernels to simulate the rich, flow record generation process. The initial PCAP file reading kernel reads in a PCAP and outputs a raw packet, which is immediately read in and processed by the raw packet parser. The raw packet parser extracts the 5-tuple from the packet and sends the 5-tuple along with the packet features as a key-value pair

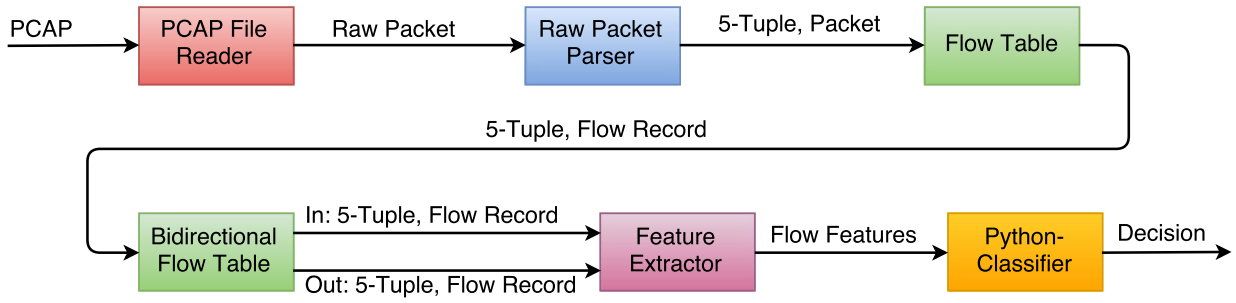


Figure 4.3: All boxes except the Python-classifier are kernels we wrote for stream processing. We built the kernels to convert a PCAP to a set of flow records for feature extraction. Each kernel executes one step in the flow processing system.

to the flow table kernel. We wrote a custom flow table to do most of the packet processing and memory management. The flow table stores a map of flow records, where the key is the 5-tuple and the value is a list of packets that are members of the flow represented by the 5-tuple. When a new incoming 5-tuple and packet arrive at the input of the flow table kernel, the kernel looks for the arriving 5-tuple in its stored flow table. If the 5-tuple is found, the incoming packet features are appended to the list of packets corresponding to the packet's 5-tuple key. If the incoming packet's key is not found, then a new entry in the flow table is created.

Unfortunately, flows are direction dependent. In a client's communication with a server, two flows are extracted. One flow corresponds to the client-to-server communication, and the other flow correlates with the server-to-client communication. In order to look at traffic burst patterns and extract other features requiring knowledge of corresponding flows in opposite directions, we wrote a bidirectional flow table kernel. Similar to the preceding flow table kernel, the bidirectional flow table manages a list of flows. However, flow records are matched with each other when an incoming flow record's source IP and source port match another flow record's destination IP and destination port and vice versa. If a match is found, the two flows are exported out of the bidirectional flow table to the next kernel. If a flow match is not found, the incoming flow is added to the bidirectional flow table and waits for a match.

After two flows are matched, they are exported to the feature extraction kernel. The feature

extraction kernel takes in both flow records and performs calculations using the features as seen in Figure 4.2. Our classifier makes use of two main types of flow features, direction independent and direction dependent. Direction independent flow features are features that do not require the knowledge of the corresponding flow traveling in the opposite direction. Flow independent features include flow duration, packet interarrival times, total number of packets and their respective lengths, and the number of unique packet lengths.

Direction dependent features are flow features that rely on knowing the features of the flow traveling in the opposite direction on the same connection. Direction dependent features include burst lengths, the ratio of outgoing to incoming packets, and the ratio of outgoing to incoming bytes. Burst lengths are defined as a sequence of outgoing packets which contain no two adjacent incoming packets. The feature extraction kernel calculates the two classes of flow features and passes them to the Python-based classifier.

4.4.2 Ransomware Classifier

As mentioned in Section 4.3.2, we tune our random forest using three main parameters: the number of trees in the forest, the depth of each tree, and the number of features used in each tree. Since the end goal is to run our classifier at line rate, we want as many trees as possible without adding significant overhead. As a result, we use 40 trees in the forest, and set the depth of each tree to 15. It should be noted that increasing the number of trees and the depth of each tree has diminishing returns. We tested numerous combinations of total decision trees and decision tree depth and found that increasing the number and depth of trees from 40 and 15 respectively resulted in minimal classification accuracy gains. Finally, due partly to convention and mainly to the high computational cost of decision tree feature splitting, we set our maximum features parameter to the square root of the total number of features in our dataset. This reduction in features greatly improves the learning time of the tree without a noticeable loss in classification performance.

4.5 Results

In this section, we present the composition of our dataset and the metrics that define success for our classifier. We also investigate the performance of our classifier in identifying ransomware as a whole. We then move on to discuss how well our classifier can identify a specific type of crypto ransomware.

4.5.1 Data Collection

We collect over 100MB of ransomware traffic traces from *malware-traffic-analysis.net*, resulting in 265 unique bidirectional ransomware-related flows. We collect another 100MB of network traffic that is malware free (clean) to use as a baseline. The clean data consists of flows corresponding to web browsing, file streaming, and file downloading. When analyzing the ransomware traffic, we analyze the traffic to and from the infected machine in communication with the C&C server. We combine both the ransomware and clean traffic and feed it into our stream processor to extract features for the classifier.

4.5.2 Success Metrics

We next discuss our success metrics, which help us determine whether or not we have produced a strong classifier. For our first success metric, we look at the recall of our classifier. The recall deals with the classifier's false negative rate. In the future, we plan to implement our system in a real-world setting to catch ransomware before it encrypts a user's computer. To do so, we need to ensure that our false negative rate is as low as possible to prevent misclassifying ransomware as clean traffic.

We next look at the false positive rate of the classifier in determining its success. The false positive rate describes how often clean traffic is misclassified as ransomware. The false positive rate also needs to be as low as possible to prevent the unwarranted blocking of clean traffic. Furthermore, a high false positive rate results in a base rate fallacy issue, which quickly results in

a massive number of falsely identified ransomware traffic.

To measure the classifier's success, we also look at the F1 score. The F1 score is a weighted average of the recall and precision scores and provides an idea of the balance between the false negative and false positive rates.

4.5.3 Feature Selection

We select our features based on the nature of the victim computer's communication with the C&C server. Since communication with the C&C server runs through multiple layers of proxy servers, we expect a higher than normal traffic latency. We extract this increased latency by measuring packet interarrival times. Furthermore, we also expect more incoming than outgoing traffic from the victim computer due to the downloading of the initial infection, the encryption key retrieval process, and the payment method notification from the Tor hidden service. We collect data to test this expectation by extracting the inflow to outflow packet ratios and burst lengths, where a burst length is the number of incoming packets before two adjacent outgoing packets are registered. The combination of interarrival times, packet ratios, and burst lengths can help distinguish a clean download from a malicious download through proxy servers.

4.5.4 Initial Classification Model

We first tune our stream processor to extract 28 unique features from our collected network traffic. These features are fed into the classifier, which first ensures the data contains the same number of malicious flows as clean flows in order to prevent classification bias. The data is then split into two, unequal sets. One set consists of 70% of the data and is used for training and the other set holds the remaining 30% of traffic and is used for testing the learned model. A 10-fold cross validation (CV) is performed on our data splitting to ensure our splitting model is unbiased. The confusion matrix in Figure 4.4 shows the results of our classifier using 28 different features. Even with a smaller set of traffic data, $\sim$200MB, we are able to achieve a respectable recall of 0.89, a precision of 0.83, and an F1 score of 0.87. If we take a look at the corresponding ROC

curve in Figure 4.7a, the area under the curve is 0.935, showing promise for successful ransomware detection. Furthermore, the average of the 10-fold CV score for our model is 0.87, indicating that we can expect similar accuracy results on other datasets.

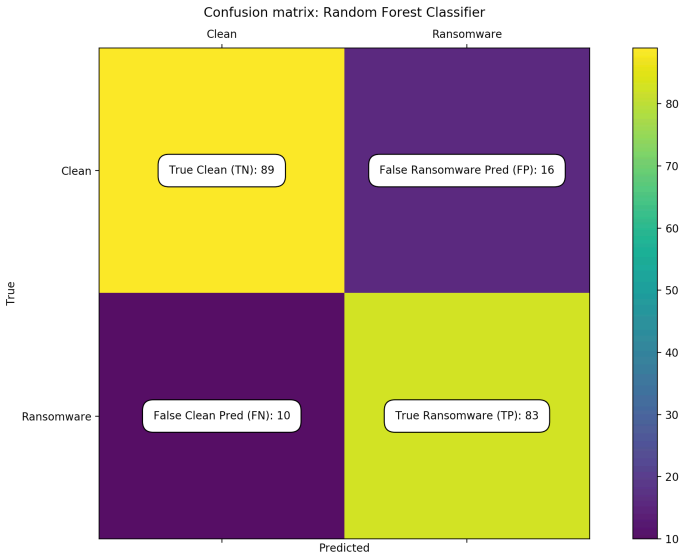


Figure 4.4: The confusion matrix of our 28-feature random forest classifier shows a recall of 0.89, a precision of 0.83, and an F1 score of 0.86.

4.5.5 Feature Reduction

Feature reduction is a key method used in machine learning to increase classification accuracy while simultaneously reducing the computational cost of the model. In order to reduce the number of feature in our model, we identify the top eight most influential features in classifying ransomware traffic, as seen in Figure 4.5. The eight features are made up of mostly inflow and outflow length and interarrival time metrics. These eight features, which are circled in red and labeled in Figure 4.5 are used to develop a new random forest model for ransomware classification.

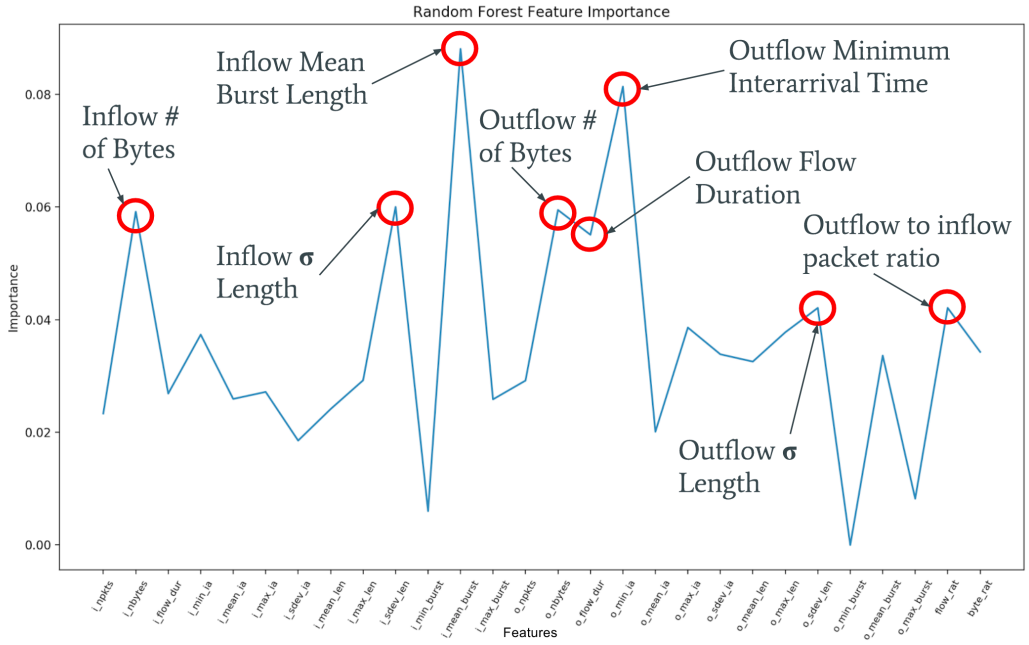


Figure 4.5: The plot above shows the weights of each of the 28 features in classifying ransomware traffic. The top 8 most important features are circled in red and labeled. We use these 8 features to train a new classifier.

After training a model using only the inflow and outflow number of bytes, inflow and outflow standard deviation of packet lengths, inflow mean burst length, outflow minimal interarrival time, and the outflow to inflow packet ratio, we test our model and produce similar results to our classifier using 28 features. The confusion matrix of our 8-feature classifier can be seen in Figure 4.6. It is clear when comparing Figures 4.4 and 4.6 that the reduction in features has little impact on the classification accuracy. The 8-feature model has a slightly lower recall score at 0.87 but produces a higher precision and F1 scores of 0.86 and 0.87, respectively. However, Figure 4.7b shows a slightly smaller AUC for the 8-feature ROC indicating that the 8-feature classifier performs about 1.4% worse than the 28-feature model. This slight performance loss will be worth the computational savings when running classification at line rate.

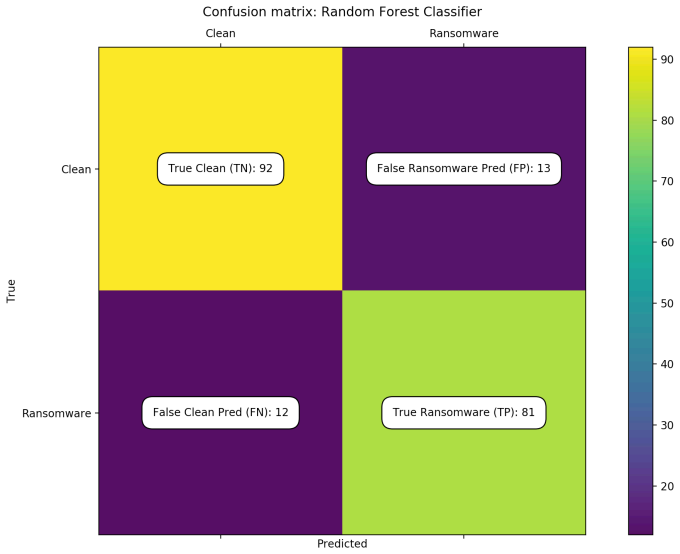
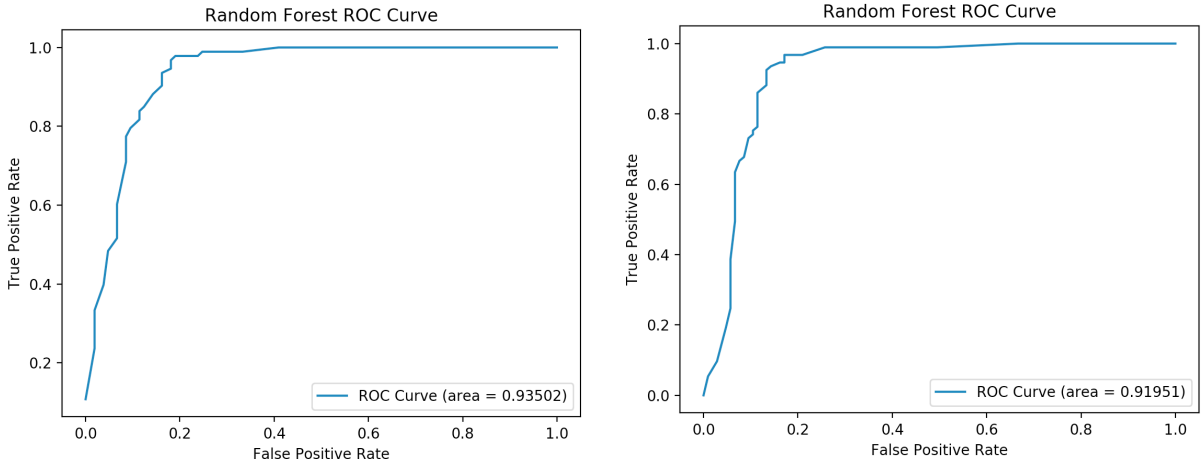


Figure 4.6: The confusion matrix of our 8-feature classifier shows similar results to that of our 28-feature classifier with a recall of 0.87, precision of 0.86, and F1 score of 0.87.



(a) 28-Feature ROC Curve. AUC: 0.93

(b) 8-Feature ROC Curve. AUC: 0.92

Figure 4.7: Comparison of ROC Curves for the 28-feature and 8-feature classifiers

4.5.6 Cerber Ransomware Detection

After running a classifier to detect all types of ransomware communication with a C&C server, we looked into specifically classifying Crypto-based Cerber ransomware, a ransomware which infected over 150,000 users in 2016 [368]. Cerber is a RaaS-type ransomware, which allows any nontechnical adversary to create and distribute their own ransomware. We chose to classify Cerber specifically due to its large infection footprint and its availability to anybody who wants to deploy ransomware.

We extract Cerber's eight most important network features, which include the mean and maximum burst lengths of the inflow stream, and create a random forest model for predicting Cerber ransomware. While we use a smaller sample size than in our previous tests, we are able to achieve a false negative rate of 0.0% and a false positive rate of 12.5%. Figure 4.8 shows the confusion matrix of the classifier. Furthermore, the ROC curve also attains a high AUC of $\sim$0.987.

It should be noted that our 10-fold CV score average comes in at 0.905, indicating that as we use the Cerber classifier on more network traffic, we are likely to see a slight rise in false negatives and false positives.

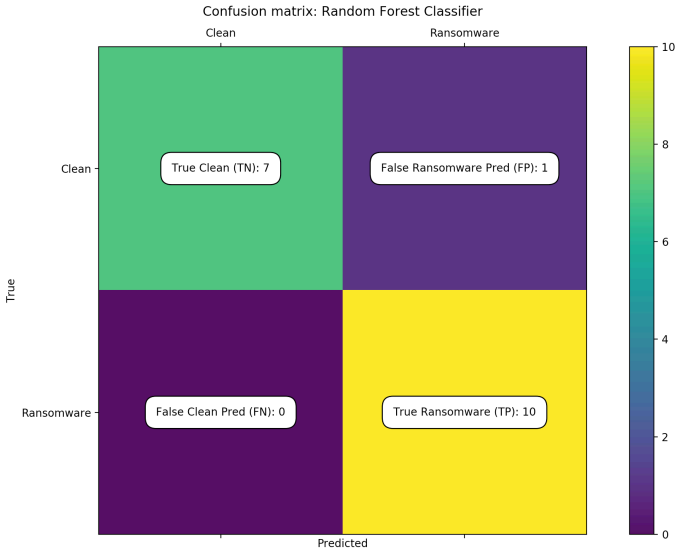


Figure 4.8: The confusion matrix of the Cerber classifier shows zero false negatives with a 12.5% false positive rate and an F1 score of 0.94. The initial findings are promising as we move forward in collecting more ransomware traffic.

While we use a small sample size for classifying Cerber traffic, the results indicate that our machine learning approach may have more success in classifying specific types of ransomware rather than ransomware as a whole. While the underlying method for distributing and launching ransomware is similar, the individual traffic shapes likely differ slightly across ransomware flavors based on the ransomware developer. We leave this investigation to future work and shift below to evaluating the quality of these network intrusion detection systems.

4.6 Towards Evaluation of NIDSs in Adversarial Setting

The work above on ransomware detection lined up with a growth in neural network-based NIDS research. Researchers have been looking at using neural networks to classify network traffic in order to differentiate between "normal" network traffic and malicious network traffic. We noticed that while moving toward deep neural networks for NIDSs holds great promise, there is an underlying problem that has yet to be addressed, their vulnerability to adversarial examples - small perturbations of the input that can bypass or purposely alter a neural network's classifica-

tion. Previous work in other domains (e.g., image classification) has shown that neural networks are vulnerable to adversarial example attacks [174, 415], small perturbations of the input that can bypass or purposely alter the classification. In the case of images, this might be changing a few pixels (imperceptible to the human eye) such that the classifier misclassifies a specific person as a different person or hides that person all together. Unfortunately, we don't fully understand the implications in the context of NIDSs because previously proposed, anomaly-based NIDSs have not been evaluated in adversarial settings [144, 446, 440, 310, 326, 457, 451, 191]. The other downside of these anomaly-based NIDSs is that they are evaluated on outdated datasets [191].

In order to address these issues, we introduced a technique in [259] to evaluate anomaly-based NIDSs in an adversarial setting. We also performed an evaluation of previously proposed NIDSs with this technique on a new dataset that contains 12 different network attacks. To do so, we needed to overcome some challenges not seen in other domains. When generating adversarial examples, we are constrained by two key factors: (i) we must retain the network protocol correctness, and (ii) we must retain the attack's semantics. We illustrated how to craft adversarial examples for networks by identifying traffic manipulations that can change the network features but remain within the constraints above. While more details can be found in our paper [259], we outline our contributions below.

First, we explained how an adversary can legitimately modify network traffic in order to fool an anomaly-based NIDS and not break underlying network protocols. We then showed how these transformations can be tailored towards a packet-based NIDS, which predicts the malicious traffic in real-time by extracting features from each packet. We demonstrated how an adversary can fool a flow-based NIDS that detects malicious traffic based on the high-level features extracted from the whole flow by considering the legitimate transformations we introduce. Finally, we evaluated the aforementioned NIDSs on a new network traffic dataset, which contains a wide range of attacks, to show how each of these attacks can be maliciously modified to fool an NIDS.

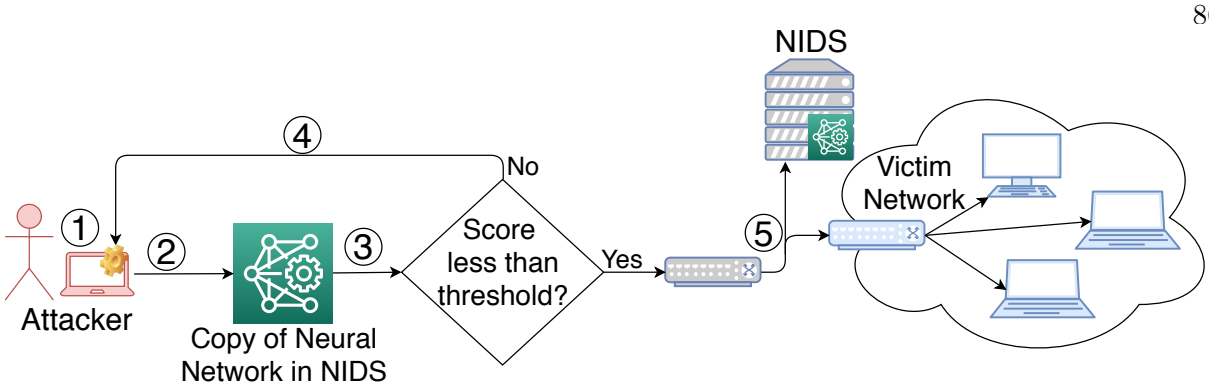


Figure 4.9: System overview and threat model considered when evaluating and designing anomaly-based intrusion detection systems. ①: The attacker sits outside the victim network and generates adversarial examples. ②: Adversarial examples are sent to the local copy of the NIDS for evaluation. ③: A classification score is produced by the NIDS based on the input. If the output score is greater than the threshold, the attacker applies some modifications, ④, to improve the adversarial example. This loop back process is carried out a maximum of `N` times. If the score in ③ is less than the threshold, the packet is mirrored to the NIDS and sent to the victim network ⑤.

4.7 NIDS in Adversarial Setting

4.7.1 Threat Model

Before outlining our approach, we define the threat model we consider in evaluating anomaly-based NIDSs. Figure 4.9 provides an overview of our threat model and system overview. In order to have a complete evaluation, we consider a white-box setting. That is to say, we consider that the attacker has a copy of the NIDS deployed on the victim network and knows all of its parameters. The NIDS deployed on the victim network receives a copy of all the packets that travel through the network entrances (⑤). We also consider that attacker's resources are limited to what she already used to create the original attack. In other words, in order to generate the adversarial version of a network attack, we assume the attacker does not want to use more machines or more IP addresses. The attacker also is considered to be outside of the victim's network (①).

4.7.2 Challenges in Crafting Adversarial Examples for NIDS

Crafting adversarial examples against NIDSs that are trained on network traffic introduces its own complications and constraints. Thus, the crafting procedure needs to be tailored for NIDSs.

Here, we mention some of the differences that exist between images and network traffic that prevent an adversary from fooling the NIDSs with the same procedure used against image classifiers. First of all, pixels in an image can be modified freely. This is not the case for a sequence of network packets. For example, if features that are fed into an NIDS are packet headers, changing some of the headers could cause the communication between the attacker and the victim to breakdown. To this extent, during the crafting procedure, attackers should ensure that the communication channel does not timeout or breakdown. Second, pixels in an image can be modified independently of each other. This is not true for typical features fed into an NIDS. In many cases these features are dependent on each other, and there is no guarantee that a valid network flow exists that matches the features generated by the crafting procedure. For example, a flow's average interarrival time between packets is directly tied to the flow's duration and number of packets through the following relationship: $Flow\_IAT_{avg} = \frac{Duration_{Flow}}{Pkt\_count_{Flow}-1}$

As a result, we cannot arbitrarily change these features independently. We must ensure the inherent properties of flows are not violated. In addition, all adversarial image pixels can be modified to fool an image classifier, but this is not the case for NIDSs. Many of the features that are fed into them are extracted from the packets generated by the victim. These are packets that the attacker doesn't have control over. The differences between adversarial image generation and adversarial network traffic generation along with the security concerns that fooling an NIDS raise, demonstrate the need to explore how an NIDS can be evaluated in an adversarial setting.

4.7.3 Legitimate Packet Transformations

If we are able to manipulate the malicious packets of an attack to have specific features that mimic benign traffic, we will be able to bypass NIDSs.

We declare an attack a success if the manipulated attack packets meet the following three requirements.

(1) The packets must carry out their original malicious intent effectively (e.g. a port scan, after transformation, should scan the victim's ports).

(2) Packet transformations must not break the underlying protocols the attack relies on (e.g. a TCP-based attack cannot violate TCP).

(3) The attack must not be flagged as an intrusion by the anomaly-based NIDSs. We will evaluate this requirement for existing systems in Section 5.6.

From these requirements and from studying the features used in existing anomaly-based NIDSs, we identify three, general, packet manipulation techniques that can be used for crafting adversarial versions of network attacks.

The manipulations are as follows:

- **Split**: The attacker can increase the number of packets sent by splitting the original payload of each packet across multiple packets. For TCP, as long as sequence numbers, acknowledgement numbers, and IP IDs are updated properly, the attack remains effective as no information is lost and the packets are reassembled at the victim host.
- **Delay**: The attacker may adjust the time between outgoing packets by either increasing or decreasing the time elapsed between subsequent packets. Since the packets themselves are not modified, the attack will not only maintain its effectiveness (so long as there is not a connection timeout), but it will also adhere to the underlying network protocols.
- **Inject**: The attacker also has the ability to construct fake packets with arbitrary lengths, transmission times, and flag combinations. She can send the decoy packets among the real attack packets as long as she can ensure that these fake packets are ignored by the victim but processed by the NIDS. By doing so, an NIDS takes into account packets that both reach and don't reach the victim into its decision on whether or not the current flow is malicious. The attacker can rely on the fundamentals of TCP, UDP, and IP protocols to guarantee these decoy packets are processed by the NIDS but not by the victim host. For example, the attacker can inject a TCP packet with a sequence number smaller than the ACK number acknowledged by the victim. Furthermore, by setting the TTL field of the IP header such that the TTL is greater than zero when processed by the NIDS but decrements to zero prior to reaching the victim, the attacker ensures the packet is dropped

after reaching the NIDS but before the victim.

Therefore, in order to fool an NIDS which is trained on network traffic packets, the adversary should modify the malicious traffic with a set of legitimate transformations as described above. In the next section, we describe how we can use these transformations to attack several NIDSs.

4.8 Crafting Adversarial Examples

In this section, we first explain how to tailor the legitimate transformations introduced in the previous section towards packet-based NIDSs, and then move on to flow-based NIDSs.

4.8.1 Adversarial Examples for Packet-based NIDSs

Algorithm 1 shows how we tailored legitimate transformations, introduced in the previous section, towards Kitsune. In a nutshell, Kitsune keeps some internal states for each flow and each packet moves through the network, updates the corresponding state. Then it calculates a score, based on features extracted from the internal state to decide whether the current packet is from a malicious traffic or not. In order to fool Kitsune, each malicious packet that is sent from the attacker, is fed through the local neural network copy and the output score is registered. If the score of that packet is close to the threshold found during training time, we see if waiting a few moments can help reduce its score. More specifically, we implement the TryDelay procedure, which performs a binary search in the range between 0 and 15 seconds to see if adding a delay can bring the score of the current packet to less than $0.9 \times threshold$. In the case that the score is greater than the threshold, we also try splitting the packet.

The TrySplit procedure tries to convert a large packet into multiple smaller packets such that the score of all of them becomes smaller than the threshold. Since we don't know what the right cut-offs are to split the original packet, we search for the correct cut-off by trying different values. More specifically, we split the payload of packet with L bytes into two packets with r and $L - r$ bytes of payload, where r is chosen randomly. Since this cutoff might not be the right one, we need to backup the state of the local NIDS related to the current flow and restore it in case

the split failed. When this happens, we try a different r. We need to do checkpoint the NIDS's state to make sure that the state of local copy remains the same as the remote NIDS. If the first portion (r bytes) of payload could fool the NIDS, we would do the same thing for the second part (the remaining $L-r$ bytes) recursively until the whole packet's payload would be sent and none of them would be detected. Finally, if delaying or splitting the original packet could help to fool the local copy, the attacker will make the appropriate change(s) and send the packet(s) to the victim. Otherwise, the original packet would be sent.

If the malicious packet is sent from the victim and its score is larger than the threshold, the only thing the attacker can do is to change the state of the NIDS such that the victim's packet do not pass the threshold. In this case, we see if injecting a fake packet from the attacker before the victim's packet can fool the NIDS for both packets such that the score of both of them becomes less than threshold. More specifically, in the TryInject procedure, we send a packet from the attacker with different payload sizes. If that packet's score is less than the threshold, we send the victim's packet after that. If the score of both packets is less than the threshold, we inject that packet, otherwise we restore the state of the local NIDS to the state before sending the fake packet. We repeat this for another fake packet with different length. Also, since the TryInject procedure is a slow process, we run it occasionally. We keep track of the times that TryInject succeeds and fails for each attack. Then, for each new packet from victim, we run the TryInject procedure with the probability of $\delta = \frac{\#successes}{(\#successes+\#failures)}$. After each success, we reset δ to one. In practice, this means that, given a network attack, if TryInject does not work for a while, we run it less frequently. If suddenly it succeeds for a packet, we again try it on consecutive victim's packets more frequently.

4.8.2 Adversarial Examples for flow-based NIDSs

In order to evaluate flow-based NIDSs in an adversarial setting, we group the features fed into them into 4 different groups. As we mentioned earlier, manipulating the features fed to an NIDS in an adversarial manner is different from changing the pixels of an image. Here we consider

Algorithm 1 Crating adversarial examples for Kitsune

```
1: procedure CraftAdvEx(x)                                  ▷ x is a malicious packet
2:     if x is sent from the attacker then
3:         if score_x > 0.9 × threshold then
4:             TryDelay(x)
5:             if score_x > threshold then
6:                 TrySplit(x)
7:             end if
8:             Send the split packets with appropriate delay if successful.
9:         end if
10:    else                                                 ▷ x is sent from victim
11:        TryInject(x)
12:        Send the fake packet before victim's packet if successful.
13:    end if
14: end procedure
```

two of the main differences. One difference is that some of the flow-based features can't be changed because the attacker doesn't have control over them since they are extracted from the victim's traffic. Also, some features depend on other features. For example, the mean of packet payloads in the forward direction can be calculated based on two other features, total length of payloads in the forward direction and the total number of forward packets. There is another type of feature in which their value depends on the actual packets of the flow and cannot be calculated by the value of other features (e.g., std of packet payloads in forward direction). As a result, we group flow features into the following four groups.

(1) Features that should not be changed because they are extracted from backward flowing packets (victim packets).

(2) Features that can be changed independently of each other by using the legitimate transformations. These include total forward packets, total number of push flags in the forward direction, maximum packet interarrival time (IAT) in the forward direction, etc.

(3) Features whose values depend on the second group and can be calculated directly by a set of them.

(4) Features that cannot be directly recalculated based on independent features, and a sequence of packets affect their values.

We tailored our adversarial crafting algorithm based on these 4 groups. We defined 3 masks that are the subset of each other. Each mask blocks a specific numbers of features from being updated by back propagating gradients through the models. The first mask only allows the procedure to modify the independent features (e.g., the second group). The second mask adds some of the 4th group features, and finally, the third mask adds all of the features of the fourth group to the set of modifiable features. In the crafting procedure, we first check whether we can fool the NIDS by using the first mask. In the case of failure we use the second and third masks. More specifically, the loss function we defined to minimize during the crafting procedure is as follows:

$$AdvLoss = F(x + \delta \odot mask_i)$$

where F is the model and $F(.)$ is the score predicted by the model. $\odot$ is the element-wise multiplication operator and δ is the perturbation that we want to find to add to the original features to fool the NIDS. By generating the adversarial features this way, we can be sure that applying legitimate transformations to the malicious flows will result in each feature from the first three groups matching the adversarial feature found.

However, the fourth group of features would have different values, and that can cause the overall flow to be detected by the NIDS. Therefore, in order to increase the chance of fooling the NIDS, in the crafting procedure, we do not stop the algorithm immediately after the score of a given sample drops below the threshold. To have a confidence interval, we continue to modify features in order to decrease the score further below the threshold. We considered this interval in order to compensate the effect of different values between the fourth group of features and increase the chance of fooling the NIDS with the real sequence of packets.

Algorithm 2 demonstrates how we tailored the crafting procedure for flow-based NIDSs. In this algorithm $threshold'$ is a smaller value than the real threshold of the NIDS to provide the confidence interval we discussed. Note that we start with a small learning rate to keep the modifications small and increase the learning rate exponentially in case of failure. The adversarial features we find with this algorithm against a given NIDS show the lower bound of the NIDSs

robustness. This is because for some of the adversarial examples, there might not be a real sequence of packets that have those features.

Algorithm 2 Crating adversarial examples for Flow-based NIDSs

```
1: procedure CraftAdvEx(x)                                  ▷ x is a malicious flow
2:     for each mask ∈ mask_1, mask_2, mask_3 do
3:         for each lr ∈ 0.001, 0.01, 0.1, 1.0 do
4:             for each i ∈ [0, totalIter] do
5:                 take one step of GD with learning rate=lr
6:                 x' ← x + δ
7:                 Recalculate group 3 features
8:                 if score_x' < threshold' then
9:                     return x'
10:                end if
11:            end for
12:        end for
13:    end for
14: end procedure
```

4.9 Evaluation

In this section, we evaluate the performance of the aforementioned NIDSs in both a normal setting and an adversarial setting with the traffic manipulations described in Section 4.7. We first discuss the dataset used, then discuss the metrics used for our evaluation and finally, empirically demonstrate to what degree Algorithms 1 and 2 are effective in fooling different NIDSs.

4.9.1 Dataset

To evaluate network intrusion detection systems, we used a highly cited dataset containing network traces of twelve network attacks from the Canadian Institute of Cybersecurity (CIC) [3] [390]. Sharafaldin et al. in [390] compared eleven available datasets based on eleven criteria and concluded that all of them have some shortages such as lack of traffic diversity and volumes, limited number of attacks, etc. Therefore they built a new dataset which satisfies all of the eleven criteria.

The attacks are: FTP-Patator, SSH-Patator, Dos slowloris, DoS slowhttptest, DoS Hulk, DoS

[3] The dataset can be downloaded at: https://www.unb.ca/cic/datasets/ids-2017.html

GoldenEye, Heartbleed, Web attacks, Infilitration, Botnet, PortScan and DDoS. These attacks were carried out over a 5-day work week in a controlled environment. Each attack was implemented using popular network tools or was written in Python by the authors. The network traces of each attack were collected to study and identify intrusion traffic characteristics.

The CICIDS2017 [390] dataset contains flows extracted from packets files using the CICFlowMeter Tool [252]. The tool also extracts 80 behavioral flow features for each flow. The full list of features can be seen in Table 4.1.

Features	**Fwd**	**Bwd**	**Flow**
Total Duration	✗	✗	✓
Total Packets	✓	✓	✗
Total Length of Packets	✓	✓	✗
Pkt Len Min/Max/Mean/Stddev	✓	✓	✓
IAT Min/Max/Mean/Sttdev	✓	✓	✓
Bytes/s	✗	✗	✓
Pkts/s	✓	✓	✓
PSH/URG Flags	✓	✓	✓
FIN/SYN/RST/ACK/CWE/ECE Flags	✗	✗	✓
Total Length of Headers	✓	✓	✗
Down/Up Ratio	N/A	N/A	✓
Avg Bytes/Bulk	✓	✓	✗
Avg Packets/Bulk	✓	✓	✗
Avg Bulk Rate	✓	✓	✗
Initial Window Bytes	✓	✓	N/A
Packets w/ payload ≥ 1	✓	✗	✗
Min. Packet Header Size	✗	✓	✗
Active Time Min/Max/Mean/Stddev	✗	✗	✓
Idle Time Min/Max/Mean/Stddev	✗	✗	✓

Table 4.1: Features extracted from flows for classifying network traffic with flow-based NIDS. ✓ and ✗ indicate whether or not the feature was calculated for packets in moving in the labeled direction. "Flow" indicates features calculated taking into account packets flowing in both directions. Features were extracted using the CICFlowMeter Tool [252].

Each flow and its corresponding flow features were labeled as either benign or with the specific attack name, but the individual packets were not labeled. Thus, in order to evaluate the packet based NIDSs, we labeled packets as malicious or benign based on the information Sharafaldin et al. provided for this dataset. From the PCAP files provided within the dataset, we excluded IPv6 packets and labeled the other packets in the following way: for each attack, we labeled all of the packets sent or received between the attacker IP(s) and the victim IP(s) as malicious for the duration of that attack. All other packets were labeled as benign. We also exclude web attacks from our evaluation because the features extracted in our evaluation are only from packet headers and detecting web attacks requires deep packet inspection. The whole dataset contains more than 56 million packets. We trained the packet and flow-based NIDSs on the Monday traffic, which contains over 11.6 million benign packets (529,481 flows). The NIDSs were then tested on the network traffic generated from Tuesday to Friday, which contains both benign and network attack traffic. This test set contains 12 different network attacks, which make up 10.33% of the overall packets and 24.22% of the overall flows. The dataset's full packet and flow statistics can be found in Table 4.2.

Set	Type	# of P	% of P	# of F	% of F
Train	Benign	11,680,917	100	529,481	100
Test	Benign	39,946,287	89.67	1,741,803	75.78
	FTP-Patator	110,736	0.25	7,935	0.35
	SSH-Patator	138,621	0.31	5,897	0.26
	DoS slowloris	47,586	0.11	5,796	0.25
	DoS slowhttptest	39,257	0.09	5,499	0.24
	DoS Hulk	2,245,526	5.04	230,124	10.01
	DoS GoldenEye	106,177	0.24	10,293	0.45
	Heartbleed	49,296	0.11	11	0.00
	Web Atks	39,823	0.09	2,179	0.10
	Infiltration	209,920	0.47	36	0.00
	Botnet	9,871	0.02	1,956	0.09
	PortScan	324,062	0.73	158,839	6.91
	DDoS	1,280,602	2.87	128,025	5.57
	All Attacks	4,601,477	10.33	556,628	24.22
	All	44,547,764	100.00	2,298,431	100.00

Table 4.2: The statistics of the dataset used for our evaluation. Columns headers containing "P" contain packet information, while column headers containing "F" show flow information.

4.9.2 Evaluation Metrics

4.9.2.1 True Positive Rate (TPR)

TPR shows the ratio of malicious traffic that is detected as malicious to all of the malicious traffic when the model's threshold is fixed to a specific number.

4.9.2.2 False Positive Rate (FPR)

FPR shows the ratio of benign traffic that is considered malicious to all of the benign traffic when the model's threshold is fixed to a specific number.

4.9.3 Performance in Adversarial Setting

In order to see how each of the aforementioned NIDSs detect adversarially modified network attacks, we chose their individual thresholds in a way to keep their FPR at 0.1 since those NIDSs can detect most of the network attacks at this rate in a normal setting. In order to evaluate Kitsune, we used GMM as its detector because it could detect malicious traffic better than using the suggested ensemble of autoencoders. To fool this NIDS, we modified the malicious packets from the CICIDS2017 dataset with Algorithm 1. We fed all the packets into the NIDS, as in the normal setting, but due to the computational complexity of crafting adversarial examples, we only ran it on the first 25,000 packets of an attack. To evaluate the flow-based NIDSs in an adversarial setting, we used Algorithm 2 to find the adversarial features for malicious flows. Due to the computational complexity of this procedure we only did it for the first 5000 flows of each attack in the cases where the attack contained more than 5000 flows.

The results of this evaluation are shown in Figure 4.10. For each NIDS considered, we show both the TPR under normal conditions, as well as under adversarial conditions. As it can be seen in this figure, for a packet-based NIDS, the detection rate drops by up to 70% (for Heartblead) in adversarial setting and for flow-based NIDSs the detection rate drops by up to 68% (for PortScan). In fact, the performance of each NIDS decreases dramatically in most cases, indicating that these NIDS are not robust in the face of adversarial examples. More specifically, for Kitsune, the average TPR in an adversarial setting across all attacks drops to 16.6% from 43.6% in a normal setting; for DAGMM, it drops to 35.2% from 60.8%, and for BiGAN-based, it drops to 35.7% from 49.3%.

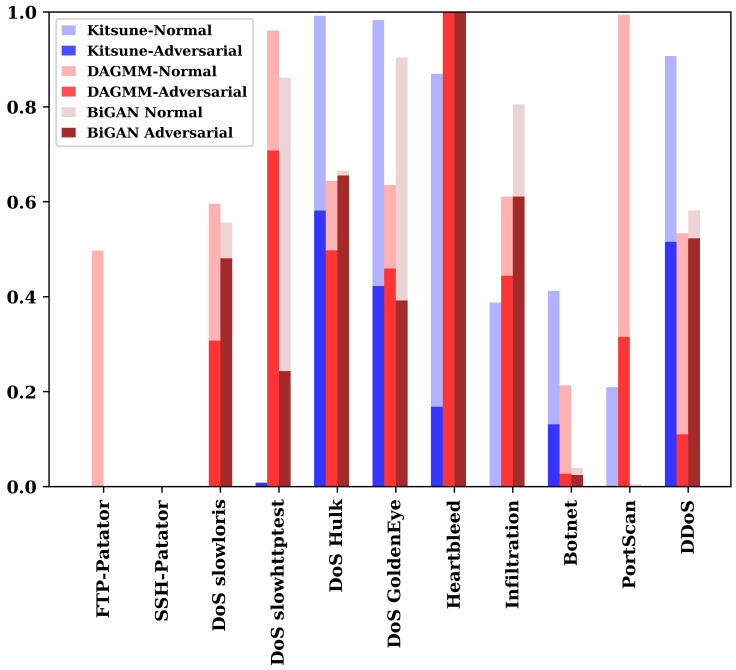


Figure 4.10: The TPR of different NIDSs for each attack when FPR is 0.1 when sending normal traffic and the adversarial version of it.

4.10 Conclusion

In this chapter, we first presented a method for detecting ransomware via its network traffic signature and then explored the efficacy of neural network-based NIDS in the face of adversarial examples. For the ransomware classification, we utilized the high processing rate of new hardware-based flow generators in combination with RaftLib's high performance and parallel framework to process rich flow records, extract flow features, and classify ransomware. Since malware communication is moving towards HTTPS for delivery and control, we only utilize the unencrypted features of HTTPS traffic for model creation. We wrote a stream processor using five kernels to process rich flow records and extract high-level flow features for use in our random forest classifier. When monitoring the communication between the infected machine and the C&C server, we were able to significantly reduce our initial feature set and achieve a detection accuracy rate of almost 87%, while maintaining a strong false negative rate close to 10%.

In our evaluation of NIDS in adversarial settings, we showed the effectiveness of our approach

by tailoring the three legitimate transformations towards both packet-based and flow-based NIDSs. We found that by using the transformations introduced in this paper, the detection rate of an NIDS trained on packet-level features can be dropped by up to 70% and the detection rate of an NIDS trained on flow-level features can be dropped by up to 68%.

Both of the above research advancements were enabled by both the performance of PFEs and the programmability and structure of the Jetstream packet processing architecture. Without PFEs and Jetstream, per packet features extraction at network speeds and scalability would be impossible. In the future, both the ransomware classifier and a more robust neural network-based NIDS could be built directly on top of Jetstream in order to process and identify network anomalies at high rates and scale.

Chapter 5

Event-driven, Sub-second Container Resource Allocation

As we've explored in the previous chapters, the rigidity of the underlying cloud infrastructure in the secure hardware and network monitoring domains prevents developers from optimizing the security, performance, and efficiency of their applications. In this chapter, we continue exploring the rigidity of cloud infrastructure in the compute domain. We first identify and evaluate the shortcomings in both application performance and efficiency when running containerized software in the cloud. We then build a new container scaling platform that pushes the limits of automated resource allocation in container environments and enables developers to optimize the performance and efficiency of their containerized applications. Recent works set container CPU and memory limits by automatically scaling containers based on past resource usage. However, these systems are heavy-weight and run on coarse-grained time scales, resulting in poor performance or efficiency when predictions are incorrect.

We propose Escra, a container orchestrator that enables fine-grained, event-based resource allocation for a single container and distributed resource allocation to manage a collection of containers. Escra performs resource allocation on sub-second intervals within and across hosts, allowing operators to cost-effectively scale resources without performance penalty. Escra is enabled by customized, kernel-level CPU telemetry and memory events that enable rapid, fine-grained scaling of containers. Escra's scaling algorithm can be customized based on a developer's application requirements, putting the developer in control over the optimization of their container's scaling decisions.

We evaluate Escra on two types of containerized applications: microservices and serverless functions. In microservice environments, fine-grained and event-based resource allocation can reduce application latency by up to 96.9% and increase throughput by up to 3.2x when compared against the current state-of-the-art. Escra can increase performance while simultaneously reducing 50th and 99th%ile CPU waste by over 10x and 3.2x, respectively. In serverless environments, Escra can reduce CPU reservations by over 2.1x and memory reservations by more than 2x while maintaining similar end-to-end performance.[1]

5.1 Introduction

Containerized infrastructure is quickly becoming a preferred method of deploying applications. The light-weight nature of containers coupled with rich orchestration systems enable a new way to design automated operations that are integrated with development workflows. In these deployments, per-container resources limits are used to prevent interference between containers and unchecked resource usage.

Setting container resource limits is a trade-off between application performance and efficient use of underlying system resources. When resource limits are set low to prioritize efficient resource use, applications will experience an increased number of CPU throttles and out-of-memory (OOM) events. Throttles slow processing and OOMs kill containers; both result in degraded application performance. When resource limits are set high to prioritize application performance, resources are underutilized which increases deployment cost [211, 186]. Developers pay the cost when cloud providers charge tenants based on resources reserved [377, 10, 27]. Cloud providers pay the cost in cases where developers are charged by usage, such as in serverless computing [22, 26, 33, 59].

Due to this trade-off, setting accurate limits is important. In practice, it is also difficult [145, 360, 377, 214, 81][2] . Using profiling to characterize application resource requirements will only result in accurate estimates if there is a representative workload. As workloads are often dynamic, the

[1] Work published at CoNEXT Companion Posters 2019 [211] and ICDCS 2022 [210]

[2] The aggregate CPU utilization at Twitter is <20% but the reservations reach up to 80%. Memory utilization is only slightly better at 40-50% but the reservations still greatly exceed the usage [214].

resources needed will change over long timescales (diurnal patterns, gradual changes in application popularity, etc.) and short timescales (bursts, failures of coupled systems, etc.). Since creating an accurate estimate of resource requirements is so complex, developers and operators often resort to over-provisioning resources. This results in underutilized deployments, a trend often observed by datacenter operators [420, 306, 249, 214, 241].

Recent work has addressed some of these challenges by leveraging machine learning to predict future needs and then automatically scaling container resource limits based on those predictions [377, 360]. These works eliminate the developer burden of setting resource limits but are constrained to using coarse-grained intervals (e.g., several minutes) to set resource limits. Coarse-grained intervals are required because the system has to learn enough information to be able to predict resource use. This is a poor fit for some workloads with short-lived containers, such as in serverless systems [388, 23, 56, 34]. Coarse-grained intervals also increase the odds of misprediction since the dynamics of applications can change throughout an interval. Thus, these works still contend with the performance and efficiency trade-off.

In this chapter, we argue the performance and efficiency trade-off can be avoided by using a **fine-grained, event-based resource allocation** scheme. To this end, we introduce **Escra**: a fine-grained, event-based resource allocation infrastructure for single containers and distributed resource allocation capable of managing resources of multiple containers across multiple nodes. We find resource allocation can easily adapt to sub-second intervals within and across hosts, allowing datacenter operators to cost-effectively scale and assign resources without performance penalty. This scheme has numerous benefits. Instead of a container being killed when it reaches an OOM event, an **event-based** system can catch the event and scale the container dynamically. Instead of making conservative allocations in order to avoid performance degradation over coarse-grained time intervals, a **fine-grained** system can always aim to right-fit allocations to current resource demands and can quickly react to instances of CPU throttling.

Escra consists of a logically centralized controller that administers resource allocations to containers across servers. Each server is instrumented with kernel hooks and runs an agent process

that applies resource decisions and reports container usage to the controller. A **Distributed Container** abstraction enforces resource isolation by enforcing per-application resource limits, similar to Resource Quotas found in other container orchestration systems [97, 95, 118, 247]. In these systems, Resource Quotas are enforced at the admission control stage. However, unlike Resource Quotas, a Distributed Container enforces resource limits both at deployment and throughout the lifetime of a container, allowing containers belonging to the same tenant to share compute resources across hosts on the order of milliseconds. Runtime limit enforcement enables Escra to fully utilize the per-application limit even when some containers are using less than their initial deployment allocation. The contributions of our work are as follows:

- We expose fine-grained telemetry data from Linux's Completely Fair Scheduler (CFS) [422]. This allows Escra to quickly track and react to actual resource needs, resulting in both high performance (low latency and high throughput) *and* low cost (minimal slack[3]).
- We implement event-based memory scaling and periodic memory reclamation. Escra uses memory scaling to increase container memory upon an OOM event, rather than allowing the container to be killed. Periodic memory reclamation increases application memory efficiency.
- We show Escra is effective by comparing slack, latency, and throughput performance to recently proposed systems. We reduce application latency by up to 96% while increasing throughput up to 3.2x over a state of the art container orchestrator. These low latency and high throughput rates are achieved while simultaneously reducing the median CPU and memory slack by over 10x and 2.5x, respectively. We show the overhead from the central controller is minimal.
- We show Escra reduces slack and both CPU and memory reservations in serverless applications without increasing application latency, potentially reducing cost to both the developer and the infrastructure provider.

[3] Slack: a container's CPU or memory limit minus its CPU or memory usage

5.2 Related Work

Current container orchestration systems (Kubernetes [71], Borg [427], Mesos [261]) set static container resource allocations. Here we present recent works that instead dynamically scale containers and discuss the limitations of these systems.

Vertical Pod Autoscaler (VPA) VPA is a Kubernetes project that implements automated container scaling through a threshold-based scaling mechanism [247]. VPA sets a target resource utilization and an upper and lower bound on that utilization. When the container usage hits the upper threshold, VPA scales the container up. When the lower bound is hit, VPA scales the container down. VPA also has the capability to enforce per-application limits via Resource Quotas [97]. A resource quota is a hard resource limit on the aggregate compute usage across all or a subset of deployments or services in a Kubernetes namespace.

Limitations of VPA VPA sets the upper and lower limit scaling bounds far apart. Since scaling a container requires a container restart, VPA only scales a container at most once per minute. The loose scaling-bound limit and infrequent container scaling results in high slack which translates to decreased cost-efficiency.

Autopilot Autopilot is a proprietary Google project that addresses the low cost-efficiency of static container deployments [377]. Autopilot runs a control loop that collects both per-second and five minute aggregated usage data from each container, analyzes it, and then makes a prediction on whether or not a container needs to be scaled. Autopilot uses machine learning predictions to scale container limits as frequently as every five minutes.

Limitations of Autopilot While Autopilot provides an automated mechanism to set limits, it does so at coarse-granularity which causes cost-efficiency and performance issues for two reasons. First, Autopilot's heavy-weight algorithm and periodic control loop prevent it from quickly responding to changes in workloads. As a result, resource predictions are forced to at least match the maximum predicted usage over the next allocation period (Autopilot uses a default 5-minute

period). This leads to unnecessary slack. Second, because Autopilot relies solely on prediction, it is unable to correct inaccurate predictions even when resources are available. Inaccurate predictions can cause unnecessary OOMs and CPU throttles.

Firm Firm also uses machine learning to improve containerized application performance and cost-efficiency [360]. While Firm does attempt to minimize CPU reservations, the primary objective of Firm is to reduce service-level objective (SLO) violations. Firm minimizes SLO violations by intelligently multiplexing compute resources to optimize the critical path of an application. Firm is similar to Autopilot because it does not require a pod restart to scale container CPU resources and can update container limits automatically.

Limitations of Firm Firm does not implement seamless or automatic **memory** scaling, requiring users to set static limits. Firm shares the limitations of Autopilot regarding performance and cost-efficiency issues as both frameworks feature a coarse-grained, ML-based feedback loop.

5.3 Introducing Escra

Escra is a container resource allocation system that achieves high performance, cost-efficiency, and strong isolation. Escra automatically scales containers in a fine-grained manner, while providing strong isolation via a new abstraction called a Distributed Container. A Distributed Container allows containers belonging to the same tenant to dynamically share resources across multiple compute nodes while capping the overall aggregate resource usage for a given application or tenant at runtime.

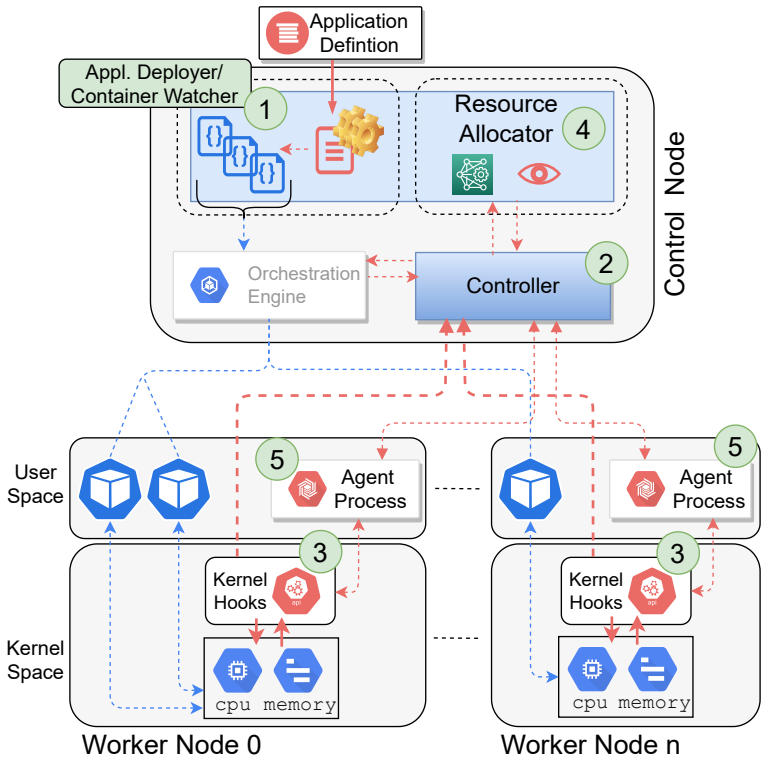


Figure 5.1: Escra Architecture. A single control node manages and controls a set of containers distributed across multiple worker nodes.

Figure 5.1 shows a high-level view of the four key components in the Escra architecture. The Application Deployer and Container Watcher (①) take a set of YAML files describing a set of Kubernetes deployments, services, and containers. The Application Deployer interfaces with the Kubernetes API to deploy containers. The Container Watcher monitors Escra containers and enables newly deployed containers to start streaming fine-grained telemetry to the Controller. The logically centralized Controller (②) handles the unique, fine-grained telemetry sent from the kernel via kernel hooks on workers (③). These kernel hooks obtain fine-grained scheduler data that is not available in user-space. A centralized controller model can be capable of scaling, as evidenced by production systems for datacenters [116] and geo-distributed network services [104]. The Resource Allocator (④) ingests telemetry from the Controller and makes per-container resource allocation decisions. Finally, similar to Kubernetes's per-node kubelet [71], an Agent is run on each host (⑤). The Agent handles resource updates sent from the Controller and can dynamically scale both CPU and memory container limits without restart on the order of 100s of microseconds. In this section,

we describe Escra's unique ability to make scaling decisions on a fine-grained timescale and in an event driven manner. A complete description of Escra's architecture follows in Section 5.4.

To illustrate the benefits of fine-grained container resource allocation, we deployed and loaded a container with sysbench [290], saturating 1-4 CPUs at any one time. The trace of the application execution with Escra is shown in Figure 5.2. Escra tracks the exact resource needs on a rapid time-scale by reacting to container throttles and OOM events and adjusting resources based on information collected during each CPU scheduling period and at OOM events. The implication of this fine-grained right-sizing is that Escra (1) significantly reduces slack and (2) simultaneously improves performance as applications are being allocated the resources they need rather than being throttled or killed due to OOMs. The remainder of this section provides further insights into how Escra achieves fine-grained resource allocation.

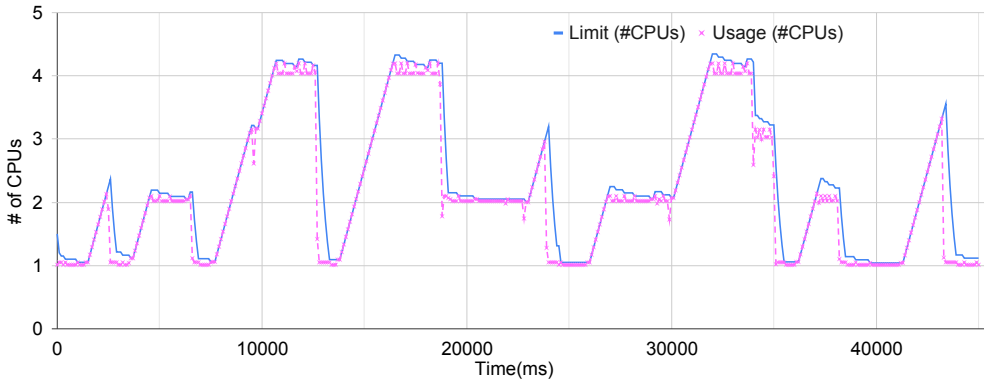


Figure 5.2: Escra's CPU tracking ability under a dynamic workload

Per-period CPU Telemetry and Dynamic Reallocation Fine-grained telemetry data is required to minimize slack via fine-grained resource allocation. Our initial analysis of systems that aggregate CPU and memory data (cAdvisor [28], Prometheus [72], Kubectl [71], etc.) found they suffer from inefficiencies stemming from reliance on coarse-grained timescales. Allocating resources quickly is not useful if allocations are based on usage data that is stale or aggregated at insufficient levels. Our goal is to obtain near-instant usage information so Escra never operates on stale data. In order to obtain fine-grained CPU data, Escra uses kernel hooks into Linux's Completely Fair

Scheduler (CFS). Upon deployment of each container, the Agent process creates a kernel socket for the container to use to report its metrics to the Controller. To implement fine-grained telemetry, containers report their per-period runtime statistics to the Controller at the end of each period. The telemetry data consists of the cgroup ID of the container, whether the container was throttled in the last period, and the amount of unused runtime in that period.

The Resource Allocator ingests raw container metrics from the Controller and uses two windowed statistics to track unused runtime and the number of throttles. The Resource Allocator uses these statistics to update per-container limits as often as every 100ms. The goal is to proactively update limits in order to keep the container limits just above container usage at all times. We update container CPU quotas using RPCs to the Agent process running on the host of the container, similar to [360].

Reactive Memory Reclamation and Reallocation upon OOM Events Escra monitors container memory usage and can seamlessly scale memory limits via two custom system calls that hook into Linux's memory cgroup structure.[4] One unique opportunity of fine-grained allocation is the ability to react to OOM events. To achieve this, a kernel hook is added in Linux's memory allocation function, `try_charge()`, to catch a container after it exceeds its memory limit and right before it gets OOMed. This hook combats inaccurate predictions within autoscalers. For example, VPA [247] and Autopilot [377] scale containers at most once a minute and once every five minutes, respectively. There is a chance a container could OOM between allocation decisions. Our kernel hook allows a container to request more memory from the Controller before the container is killed. While this is a reactive mechanism for memory scaling, the request lookup penalty is orders of magnitude faster than a container crash and restart.

One beneficial aspect of this OOM-preventing kernel event is the Resource Allocator can determine how to allocate additional memory resources depending on the state of the node and the application. If there is available memory on the node, the Allocator can simply scale the needy container up. If the node is under memory pressure, the Controller can launch an aggressive

[4] Docker supports seamless container scaling [40], but Kubernetes does not.

memory reclamation process that reclaims memory from other containers on the node with high slack. Not only will this free up memory for the container in need, but it also increases node utilization, reduces slack, and improves cost-efficiency.

Proactive Periodic Memory Reclamation In order to reduce memory slack, the Escra Controller periodically contacts the Escra Agent on each worker node, asking the Agent to reduce the memory limits of each container on the same node as the Agent. The Agent checks the usage and the limit of each container it manages. If the limit of a container exceeds the usage of the container by more than δ bytes, then the Agent shrinks the container memory limit such that the memory limit minus the memory usage equals δ bytes. Each Agent then reports back the total reclaimed memory from its containers to the Escra Controller. The Resource Allocator can then give the reclaimed memory to other containers experiencing memory pressure.

5.4 Escra Architecture

This section describes the architecture of Escra, our container orchestrator built with Kubernetes, that implements (i) automated container limit settings, (ii) seamless container scaling, (iii) fine-grained resource allocation, and (iv) dynamic, per-tenant resource sharing and collective resource limits enforced at runtime. Escra implements these features using fine-grained telemetry, event-based memory scaling, aggregated application-wide resource limits, and a centralized Controller and Resource Allocator.

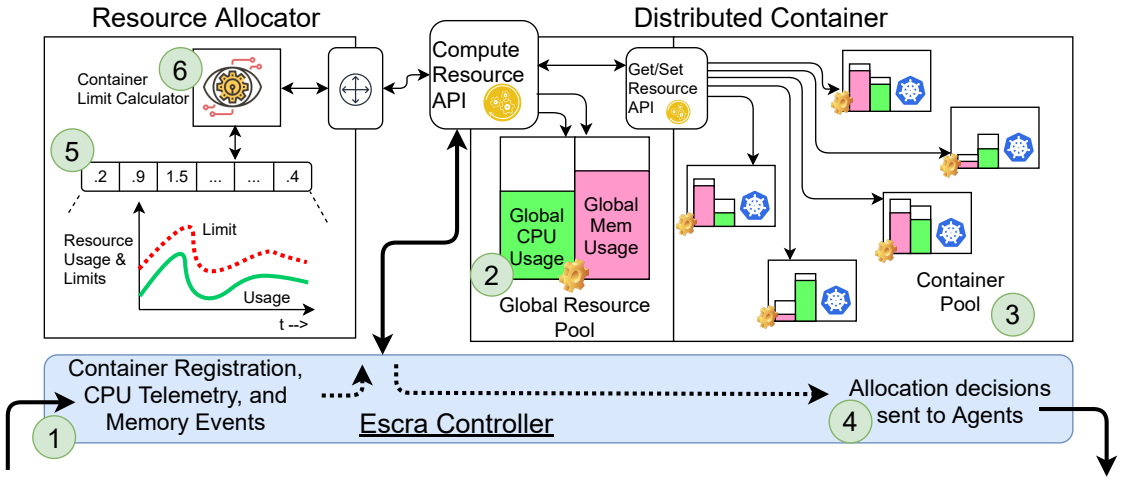


Figure 5.3: Escra Controller, Resource Allocator, and Distributed Container

5.4.1 Application Deployer & Container Watcher

The Application Deployer ingests a Distributed Container configuration as a set of YAML files (Figure 5.1, ①) describing a set of containers, and maximum CPU and memory limits. The maximum CPU and memory limits represent the limit on the aggregate usage of all containers in the application (Figure 5.3, ②). Prior to deploying the containers via Kubernetes, the Deployer sends the global application limits to the Controller. This informs the Resource Allocator (Figure 5.1, ④) of the total maximum usage of the containers in the deployment. Once the Deployer sends the application limits to the Controller, the Controller is ready to accept network connections from each container.

Initial limits are set to bootstrap containers when they first deploy but these limits will be changed by the Controller at runtime. The Deployer initializes the CPU and memory limit of each container to:

$$\frac{global\_cpu\_limit}{\#\ containers} \quad (5.1) \qquad \frac{global\_mem\_limit * \sigma}{\#\ containers} \quad (5.2)$$

where σ is a configurable parameter representing the percentage of the global application memory limit to be withheld for containers that experience OOM events.

The Container Watcher integrates with Kubernetes to detect container creation. Upon detection, the Watcher notifies the Agent (Figure 5.1, ⑤) located on the same host as the newly

created container.

5.4.2 Kernel Hooks

Escra uses kernel hooks to enable fine-grained telemetry and trap OOMs. After an Agent is notified that a new container has deployed, the Agent invokes a custom syscall that carries out three tasks, each implemented via kernel hooks (Figure 5.1, ③). First, the syscall creates a TCP kernel socket to message the Controller (Figure 5.1, ②) and informs the Controller of the existence of the container. The per-container TCP kernel socket will persist for the life of the container. Once the Controller registers the new container, it updates the container's CPU and memory limit based on the global application limits and current application resource use.

Next, the syscall modifies the container's underlying Linux CPU and memory cgroup structures to enable fine-grained telemetry and event handling. For CPU, the syscall hooks into Linux's Completely Fair Scheduler to extract runtime data to stream to the Controller. At the end of each period, the hook writes the container's cgroup quota, unused runtime (the `runtime` variable in the CFS Bandwidth kernel structure), and whether the container was throttled in the last period into a shared FIFO buffer in the kernel[5] .

After the hook finishes writing data to the buffer, the runtime of the cgroup is refilled and the next period begins. Per-container kernel threads consume statistics from the FIFO queue and send the queued CPU statistics over UDP to the Controller. Along with the container quota and remaining runtime, the CPU statistic message also includes a tag letting the Controller know what container the incoming statistic refers to. The hook will report statistics once per-period for the life of the container.

To handle OOM events, the syscall adds a kernel hook in the memory cgroup structure (`mem_cgroup`) for the container. If a container exceeds its memory limit, before it is killed this kernel hook forwards the OOM event to the Controller over the existing TCP kernel socket that

[5] Note that per-period unused runtime is not available in userspace and while one could interpret similar data from the `cpuacct` cgroup subsystem, `cpuacct` was never designed for accuracy and was initially designed as a way to showcase the capabilities of cgroups [181].

was previously used during container initialization. If memory is available in the global application pool, the container can increase its memory limit and continue running.

5.4.3 Controller

The Controller brings all of the system components together and coordinates their interactions. Figure 5.3 shows a more detailed view of the Controller, Resource Allocator, and the Distributed Container abstraction.

When containers register themselves with the Controller upon deployment, the Controller creates a logical container object and adds it to a pool of the other Escra containers within the application (Figure 5.3, ②). The logical pool of Escra containers is used to maintain an updated view and status (resource usage, limit, etc.) of the containers it is managing.

Once all containers are deployed and registered with the Controller, the Controller becomes responsible for several additional tasks. The Controller is responsible for launching a periodic memory reclamation process, handling fine-grained telemetry data from all containers, and handling memory requests from containers under memory pressure (Figure 5.3, ①). The Controller is also responsible for carrying out allocation decisions made by the Resource Allocator (Figure 5.3, ④). The Controller is *not* responsible for making those CPU and memory allocation decisions.

The Controller launches a periodic reclamation loop on behalf of the Resource Allocator that triggers each Agent to reclaim excess reserved but unused memory from each container in the cluster. The Resource Allocator determines to what extent each container's memory is resized. Every 5 seconds, the Controller sends a request to each Escra Agent, requesting the Agent to reduce the memory limit of each Escra container, $C(i)$, and send back the amount the container was resized by ψ. This resized value is the amount of memory reclaimed from that specific container. The reclaim process is as follows. The Agent reduces the memory limit on a container if:

$$C(i)_l > C(i)_u + \delta$$

where $C(i)_l$ and $C(i)_u$ are the memory limit and usage of the container, respectively, and δ is

a tunable parameter managed and set by the Resource Allocator that represents the memory reclamation "safe margin." If the condition above is satisfied, the container limit is updated via: $C(i)'_l \leftarrow C(i)_u + \delta$, otherwise, the container limit is left unchanged. We empirically set the safe margin to 50 MiB. The amount of reclaimed memory is measured as:

$$\psi \leftarrow C(i)_l - C(i)'_l$$

where $C(i)'_l$ is the resized container limit and ψ is the amount of reclaimed memory. Therefore, for each container that is resized, the Agent passes back to the Controller ψ bytes of memory. The Escra Controller forwards ψ bytes to the Resource Allocator which then adds ψ bytes of memory into the global memory pool via: $global\_mem\_limit \leftarrow global\_mem\_limit + \psi$. Note that the Controller passes all CPU telemetry data, memory requests, and reclaimed memory updates to the Resource Allocator.

5.4.4 Resource Allocator

The Resource Allocator is the lightweight decision-making component that determines the containers whose resources should be allocated to or reclaimed from. The Resource Allocator is composed of three key components. First, it has a global resource pool for both CPU and memory. For CPU and memory, it keeps track of the maximum application limit (Figure 5.3, ②), the total allocated resources, and the total unallocated (or available) resources (Figure 5.3, ⑥). Second, the Resource Allocator collects fine-grained CPU telemetry data from the Controller and uses a lightweight algorithm to make decisions on whether or not to scale up or scale down individual container CPU limits (Figure 5.3, ⑤). Third, the Resource Allocator consumes *out-of-memory* events sent from the Controller and, based on the globally available memory, increases the memory limit of memory-pressured containers.

If a container is not using up to its allocated resource limit, the Resource Allocator will trigger the Controller to take away those excess resources. However, the Allocator is designed to quickly identify when resources need to be given back to containers and will instruct the Controller to update container limits as needed.

5.4.4.1 Dynamic CPU Allocation

The CPU allocation algorithm consumes CPU telemetry data sent from each container across all nodes in order to share CPU allocations across nodes and remain under the maximum CPU limit (Ω_l). At the end of the container running period t, the Resource Allocator consumes a runtime statistic from the Controller. The runtime statistic for a container i during period t ($C(i)[t]$) includes the container quota ($C(i)_q[t]$) in ms, the amount of unused runtime ($C(i)_q[t] - C(i)_u[t]$) in ms, and whether the container was throttled ($C(i)_{th}[t]$) in the last period t.

The Resource Allocator uses two sliding windowed statistics that track (i) the excess runtime a container has at the end of each period and (ii) if a container was throttled during the last period. Based on these windowed statistics, the Resource Allocator determines whether a container needs or has excess CPU runtime and updates container quotas. A container quota (or limit) during period t is increased if $C(i)_{th}[t] = 1$ and will be increased for the following period $t+1$ via:

$$C(i)_q[t+1] = C(i)_q[t] + \frac{\sum_{t=0}^{n} C(i)_{th}[t]}{n} * \Upsilon(\Omega_l - \sum_{i=0}^{\lambda} C(i)_q[t])$$

where $\frac{\sum_{t=0}^{n} C(i)_{th}[t]}{n}$ is the windowed statistic measuring the average number of throttles over the last n container periods, $\sum_{i=0}^{\lambda} C(i)_q[t]$ is the unallocated CPU runtime for the entire application, λ is the number of containers in the application, and Υ is a tunable parameter that affects the rate at which a container CPU quota is scaled.

A container quota during period t is decreased if $C(i)_q[t] - C(i)_u[t] > \gamma$, where γ is a tunable parameter that adjusts when container quotas should be scaled down. A container quota for period $t+1$ is scaled down via:

$$C(i)_q[t+1] = C(i)_q[t] - \kappa \frac{\sum_{t=0}^{n} (C(i)_q[t] - C(i)_u[t])}{n}$$

where $\frac{\sum_{t=0}^{n} (C(i)_q[t] - C(i)_u[t])}{n}$ is the windowed statistic measuring the average runtime remaining during the last n container periods, and κ is a tunable parameter that affects the rate at which

container quotas are scaled down. We empirically found that systems with high variance in CPU usage between periods performed better with a larger Υ and a smaller γ and κ.

5.4.4.2 Dynamic Memory Allocation

This section details the Resource Allocator algorithm for handling *out-of-memory* events received from containers and ensuring the proper sharing of memory resources across an application. The Resource Allocator determines the amount of additional memory to allocate to containers under memory pressure and the amount of memory to reclaim from containers with unused memory.

The Resource Allocator consumes *out-of-memory* events that are sent from a container just before the container is killed for exceeding its memory limit. Upon receiving an *out-of-memory* event from a container $C(i)$, the Resource Allocator checks if there is unallocated memory available in the global resource pool. If there is no available memory (all global memory has been allocated to containers), the Allocator tells the Controller to reclaim unused memory from other containers in the application (described in Section 5.4.3). We implement *out-of-memory* events in Escra this way to avoid killing a container for exceeding its memory limit when available memory in the application exists.

If the Controller is able to reclaim memory from other containers in the application, the Resource Allocator will allocate a fixed number pages of memory to $C(i)$ by invoking the Agent to update the memory limit of $C(i)$. If the Allocator is unable to reclaim any memory from other containers, $C(i)$ is killed by the operating system (as is standard).

5.4.5 Integrating Escra With Serverless Frameworks

The fine-grained approach to resource allocation in Escra is well suited to serverless environments due to the high degree of multitenancy in serverless systems as well as the short-lived nature of serverless functions. Since functions have short execution times (90% execute in under 1 minute [388]), coarse-grained resource management solutions are insufficient for serverless workloads. Since Escra is fine-grained and designed for use with containers, it is compatible with

serverless frameworks that use containers to isolate serverless functions.

We choose OpenWhisk [16], an open-source serverless platform, as an example to illustrate how Escra may be integrated with serverless frameworks. In our configuration, OpenWhisk is deployed via Kubernetes and serverless functions (termed **user actions**) are run in pods. Each pod is deployed as part of the Kubernetes `openwhisk` namespace. Treating OpenWhisk as a single application, one can use the `openwhisk` namespace and invoker `containerPool` memory limit to set global application memory in Escra. We modified pod affinity to ensure OpenWhisk infrastructure was deployed on dedicated infrastructure nodes so there would be no resource contention between architectural components and user actions. While there is no global invoker CPU limit in OpenWhisk, one can set memory and CPU to scale linearly, which indirectly sets a global CPU limit. Escra does not delay container creation in OpenWhisk because the connection between a container and the Controller does not block the container from beginning to execute. Escra already interfaces with Kubernetes so no further modifications are needed for a minimal integration that allows all user action pods to benefit from resource sharing and reclamation.

5.5 Implementation

Escra implementation consists of a total of 14.1k SLOC. The Controller and Resource Allocator are written in C++ and utilize gRPC to communicate with the Deployer, Watcher, and Agents (all written in Go). The Deployer sits on top of Kubernetes and integrates with the Kubernetes deployer API via client-go [32] to deploy Escra containers. Docker is used as the underlying container runtime. The Container Watcher integrates with the Kubernetes work-queue API and communicates with the Agent via gRPC as well.

Escra worker nodes run a custom Linux kernel based on Linux kernel 4.20.16. The custom kernel includes a hook in the CFS cgroup subsystem and in the memory management subsystem. The kernel also includes a custom message structure used for CPU telemetry reporting and memory requests to the Controller. The rest of the kernel modifications include approximately 1,500 SLOC spread across six kernel modules that implement limit resizing and CPU telemetry.

5.6 Evaluation

The goal of Escra is to automatically and seamlessly achieve high performance, cost-efficiency, and isolation. As fine-grained allocation is a key capability of Escra, the first goal of our evaluation is to show how much Escra's highly reactive decision making process is able to improve both performance and cost-efficiency in comparison to common practice (static allocation) and a state-of-the-art system (Autopilot). Our second goal is to show how Escra can reduce the overall reservation requirements for serverless applications, while maintaining application performance; this has the potential to reduce cost for both the application owner and the infrastructure provider.

5.6.1 Experimental Setup

Experiment clusters are created using Cloudlab [221] resources consisting of a control node and worker nodes. Along with the default Kubernetes components, the control node runs the Escra Deployer, Watcher, Controller, and Resource Allocator. Each worker node runs an instance of the Escra Agent.

Microservice Benchmark Applications We first evaluate Escra on a set of four microservice applications running across three worker nodes and one control node. Each node consists of two Intel Xeon Silver 4114 10-core 2.20 GHz CPUs, 192GB of ECC DDR4-2666 memory, and a dual-port Intel X520-DA2 10Gb NIC. We set κ to 0.8, γ to 0.2, and Υ to 20 in the Resource Allocator for all experiments unless otherwise stated.

The microservice applications represent a set of four interactive, real-world benchmarks: (1) *MediaMicroservice* [237] (32 containers): a microservice similar to IMDB [60] where users can search, review, rate, and add films, (2) *HipsterShop* [55] (11 containers): an online shopping microservice consisting of standard browsing and purchasing of various items, (3) *TrainTicket* [109] (68 containers): a microservice that simulates a train ticket booking service consisting of searching, booking, modifying tickets, and (4) *Teastore* [103] (7 containers): a simulated online tea store where users can browse and purchase hundreds of various teas.

For each microservice experiment we load the microservice with one of four workload distributions: a fixed request rate, an exponentially distributed request rate, a bursting request rate, and an Alibaba datacenter trace [7]. The Fixed workload sends requests at a constant 400 requests per second. The Exponential (Exp) workload sends requests in an exponential distribution with $\lambda = 300$. The Burst workload sends a fixed 50 req/sec. with an additional 10 second exponential burst of requests where $\lambda = 600$ every 20 seconds. Finally, the Alibaba workload is sped up by 10x and sends requests at rates anywhere from 56-548 req/sec.

Evaluation Metrics Below is a list of metrics used in this section (derived from [377]):

- **Absolute Slack**: The container CPU or memory limit minus the container CPU or memory usage.
- **Application Throughput**: Measured in successful requests per sec.
- **Application 99.9%ile Latency**: Measured as the 99.9%ile end-to-end latency.

Autopilot Implementation Autopilot [377] is not open-source so we implemented a recreation of the Autopilot ML recommender to compare against Escra. The Autopilot ML recommender is inspired by a multi-armed bandit problem in which an agent tries to use the best set of arms to maximize the total reward gain over time. Some parameters used in the Autopilot algorithm are manually tuned by their engineers (w_o, w_u, etc.). As they did not specify what values they used for these parameters, we tuned them to values that resulted in the best performance.

Note that Autopilot defaults to updating container limits every 5 minutes. We tested the update period of Autopilot at 60, 30, 10, and 1 seconds and saw finer-grained update periods achieve better performance. The throughput of HipsterShop with Autopilot at 1, 10, 30, and 60 second update periods degrades from 422 req/sec. to 382 req/sec. to 279 req/sec. to 108 req/sec., respectively. While we do not know how practical it is to run Autopilot at that granularity at scale, we show comparisons against 1 second intervals as a best case for Autopilot.

5.6.2 Performance - Cost-Efficiency Trade-off

Intuitively, there exists a resource allocation trade-off between performance and cost-efficiency. One can allocate a large amount of resources to eliminate any possible performance penalty (measured in throughput and latency), but this leads to poor cost-efficiency (measured in terms of slack). In contrast, one can significantly under-allocate resources and improve the cost-efficiency, but this is at the price of reduced performance. We further examine this trade-off in the context of both common practice (static allocation) and state-of-the-art (Autopilot), and illustrate that Escra achieves better performance and cost-efficiency than each system, and that the other systems compromised on one of the metrics.

First, we estimated the resources needed for the MediaMicroservice from the Deathstar Benchmark [237] by profiling each container and measuring maximum CPU and memory usage. We then ran the application in underutilized (limits set at 0.75x the profiled max), best-estimate (set at 1.0x), and safe buffer (set at 1.5x) cases. For each case, we measure the end-to-end performance (latency and throughput) and slack (CPU cores allocated minus cores used, and MiBs allocated minus MiBs used). As expected, performance increased (i.e., latency decreased and throughput increased) with more resources allocated; however, slack (resource wastage) also increased. We find the 1.5x allocation level illustrates a sufficient buffer and use that setting for evaluating the trade-offs in comparison to Autopilot and Escra.

For this evaluation, we deployed each microservice and used the workload generation-based benchmarking tool wrk2 [419] with the four different workloads. Each application is evaluated when managed by 1.5x static limits, Autopilot, and Escra. This setup allows us to measure both latency and throughput to quantify the performance in each approach, and slack to quantify the cost-efficiency of each approach. Figure 5.4 shows the resulting *change* in latency and *change* in throughput between Autopilot and Escra and between static limits and Escra for all four applications and workload distributions. Table 5.1 summarizes our results and is broken down in the subsequent sub-sections.

App Comp.	Avg. Δ Latency	Avg. Δ Tput.	Avg. Δ 50% CPU Slack	Avg. Δ 99% CPU Slack	Avg. Δ 50% Mem. Slack	Avg. Δ 99% Mem. Slack
Static vs. Escra	38.0%	25.4%	81.3%	74.2%	55.0%	95.9%
Autopilot vs. Escra	36.1%	54.5%	78.3%	78.6%	26.7%	68.9%

Table 5.1: Average performance increase and average slack reduction for both CPU and memory between static and Escra and between Autopilot and Escra. Escra improves performance, while significantly reducing slack

5.6.3 Static Allocation vs. Escra

We first look at the change in both latency and throughput between a statically allocated application and an application deployed with Escra. Table 5.1 show that on average, Escra decreases latency by 38% and increases throughput by 25.4% compared to statically allocated applications. Escra can achieve these performance numbers with an average 50%ile and 99%ile CPU slack improvement of 81.3% and 74.2%, respectively. Escra also decreases 50%ile and 99%ile memory slack by 55% and 95.9%, respectively.

In an ideal world, we would not see a performance improvement from Escra over a statically deployed application allocated 1.5 times the peak measured resource usage; the static deployment would never experience any throttles or OOMs. However, this result is a testament to how difficult it is for developers to set resource limits on containers [145, 360, 377, 214, 81]. Not only is it hard to profile containers, since you never know what the workload rate is truly going to be, but also the tools to measure resource usages (especially for CPU) tend to aggregate over seconds to minutes, smoothing out usage spikes [28, 72, 89].

The other reason for the performance difference between Static Allocation and Escra is from the fact that Escra can dynamically share and shift resources between containers at runtime. For example, in a static deployment, when a container is underutilized (C_u) and another container is getting throttled (C_t), C_t cannot use any of C_u's resources. However, in Escra C_u is scaled down while C_t is scaled up (without exceeding the per-application global limit). Escra's ability to shift resources among containers and enforce a per-application limit at runtime, enables an application to fully utilize its allocated CPU and memory. This is a Distributed Container's main difference to

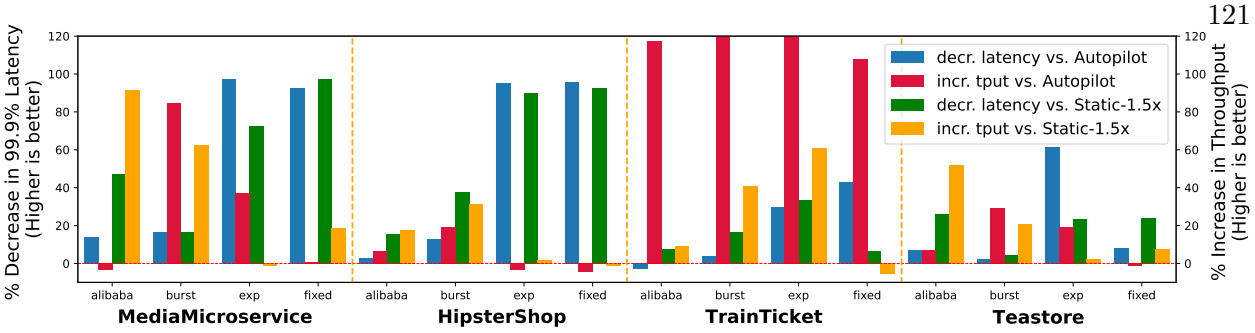
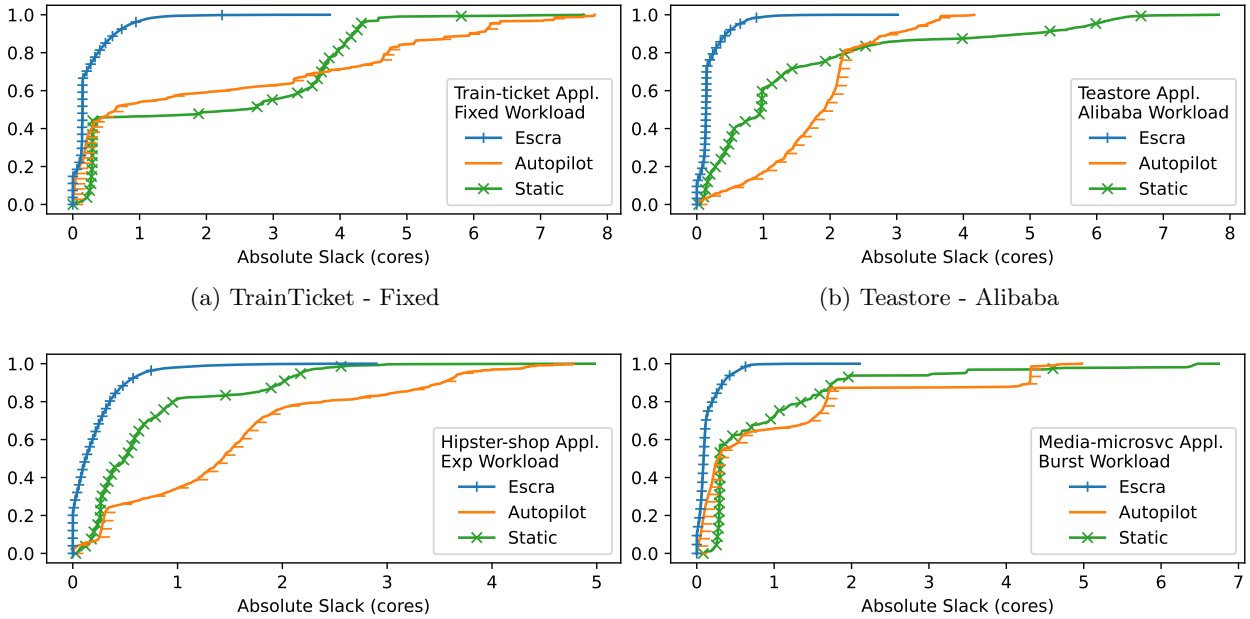


Figure 5.4: Change in 99.9% latency and throughput between Autopilot, the 1.5x measured peak static allocation and Escra. Note: TrainTicket with Burst and Exp workloads experienced a throughput increase of 134% and 324% respectively but are cut off at the top of the figure



(a) TrainTicket - Fixed

(b) Teastore - Alibaba

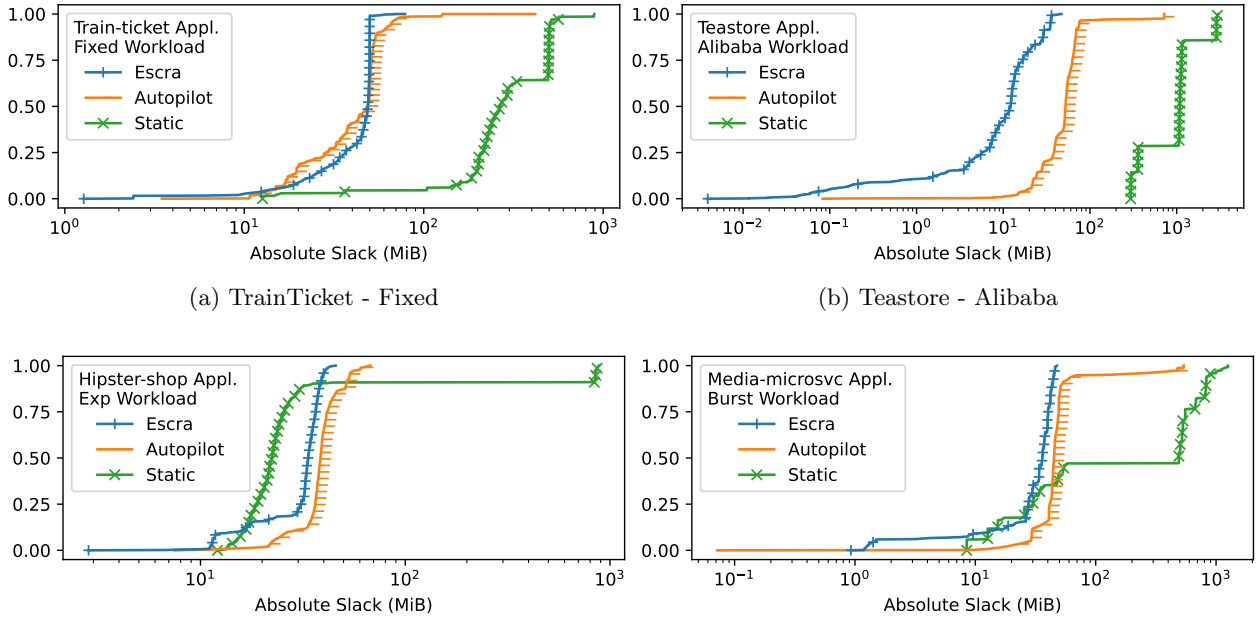
(c) HipsterShop - Exp

(d) MediaMicroservice - Burst

Figure 5.5: CPU slack CDFs comparing Escra, Autopilot, and statically deployed resources across the MediaMicroservice, HipsterShop, TrainTicket, and Teastore microservices with various workloads

Resource Quotas [97, 95]. Resource Quotas are only enforced at container deploy time, so in the case above, C_t cannot scale up because C_u is already deployed and the global limits were enforced on deployment. In the case of VPA [247] (discussed in Section 5.2), the autoscaler would have to constantly kill and restart containers as CPU usages changed.

We break down TrainTicket with Fixed and Teastore with Alibaba experiments in the fol-



(a) TrainTicket - Fixed

(b) Teastore - Alibaba

(c) HipsterShop - Exp

(d) MediaMicroservice - Burst

Figure 5.6: Memory slack CDFs comparing Escra, Autopilot, and statically deployed resources across the MediaMicroservice, HipsterShop, TrainTicket, and Teastore microservices with various workloads. The x-axis is log scale

lowing paragraphs to help illustrate the ability of Escra to achieve both high performance and cost efficiency.

TrainTicket with Fixed Workload Figure 5.4 shows that TrainTicket with Fixed performs slightly worse with Escra than with static allocation, seeing a 5.5% decrease in throughout. Examining the slack in Figures 5.5a and 5.6a, 50% of the time, the static allocation has over 2.5 cores of CPU slack and 256MiB of memory slack. In contrast, Escra has a 50% CPU slack of 0.14 cores (a 17.9x improvement) and memory slack of 49MiB. This experiment shows the trade-off the static deployment makes, sacrificing significant cost-efficiency for a slight performance increase.

Teastore with Alibaba Workload Escra improves latency and throughput of Teastore by 25.7% and 51.6%, respectively. Figures 5.5b and 5.6b show while Escra is able to increase performance, it can do so while reducing 50%ile and 99%ile CPU slack by over 81% and 74% respectively, while also significantly reducing memory slack.

5.6.4 Autopilot vs. Escra

Autopilot aims to reduce slack without sacrificing performance using ML. However, Table 5.1 shows on average, Escra decreases latency by 36.1% and increases throughput by 54.5% compared to Autopilot. Table 5.1 also shows Escra's average 50%ile and 99%ile CPU slack improvement over Autopilot is 78.3% and 78.6%, respectively. Escra also decreases 50%ile and 99%ile memory slack by 26.7% and 68.9%, respectively. We further examine the results of two of these experiments below to determine how Escra can achieve both high performance and high cost efficiency.

HipsterShop with Exp Workload In a few cases, Autopilot gets some performance improvements over Escra since it trades for performance gains at the cost of slack. Autopilot increases the throughput of HipsterShop compared to Escra by 3.16%. However, Figures 5.5c and 5.6c show Autopilot over allocates resources, with the median slack greater than 1.43 cores and 20% of allocations over 2.38 cores. For Escra, the median slack is 0.12 cores (an 11.6x decrease) with an 80%ile CPU slack of 0.35 cores.

MediaMicroservice with Burst Workload Figure 5.4 shows Autopilot degrades MediaMicroservice with Burst throughput and increases its latency. This indicates that Autopilot fails to quickly react to rapid and significant changes in CPU workloads and memory usages, resulting in low slack but higher latency and lower throughput. For the same application and workload, Escra is able to not only increase latency and throughput performance by 16.6% and 84.3%, but also able to reduce slack over Autopilot. Escra has a 99%ile slack less than 66% of a core and a 99%ile memory slack of 46MiB.

5.6.5 Takeaways

Table 5.1, Figure 5.4, and the four cases above show Escra rarely performs worse than static allocation and Autopilot, but when it does, the performance degradation is small and the slack savings are significant. When Escra outperforms the static allocation and Autopilot, Escra does so with significantly reduced slack, proving that Escra is able to achieve both high performance and

high cost efficiency. One of the key reasons for the high performance Escra is that Escra is able to greatly reduce OOMs. In all 32 experiments, Escra experienced zero OOMs, while Autopilot had up to 8 OOMs in a single experiment.

5.6.6 Serverless

This section shows how Escra integrates with OpenWhisk [16] by benchmarking two applications: ImageProcess and GridSearch. We run ImageProcess with one control node, three worker nodes, and two nodes reserved for serverless infrastructure (i.e., OpenWhisk and and a data store). The GridSearch application runs with one additional worker node. Each node is composed of two Xeon E5-2650v2 8-core 2.6 Ghz CPUs, 64GB of DDR-3 memory, and a dual-port Intel X520 10Gb NIC. For both applications OpenWhisk is configured to create each user action pod with 1 vCPU for CPU request and limit, and 256 MiB of memory. We set κ to 0.8 and γ to 0.2 for both applications and Υ to 35 for ImageProcess and 20 for GridSearch in the Resource Allocator.

Serverless Benchmark Applications ImageProcess is a single-function application inspired by the image processing application in [445]. The function reads an image from a database, processes image metadata, creates a thumbnail, and writes the thumbnail to the database. Our workload is simple: an ImageProcess request is sent every 0.8 seconds over 10 minutes. We perform four iterations of the experiment for a total of 3k invocations for each test case. At the beginning of each experiment, we ensure there are no ImageProcess pods running (to ensure initial cold starts).

GridSearch is a traditional approach for tuning hyperparameters in classifiers. This batch-like application [58] uses ˜115 serverless function pods to classify an Amazon product review dataset using scikit-learn [98] and tunes the classifier hyperparameters using the GridSearch algorithm. Each function is charged with completing tasks until all 960 tasks are completed. GridSearch uses the Lithops framework [77] for orchestration. We set the Lithops serverless backend to OpenWhisk and the Lithop storage backed to Redis.

The reason Υ is set to different values for GridSearch verses ImageProcess is due to the differences in workload characteristics. In GridSearch, each user action is relatively long-lived as

each action is a worker that will complete as many tasks as possible. Thus, it was performant to give Υ for GridSearch the same value used for microservices. In ImageProcess, a user action is a short-lived request. As such, container reuse is common and containers may experience periods of idleness between user actions. Increasing Υ allows containers to more quickly be granted the resources they need as they are created and as they transition from idle (unused) to used (running a user action).

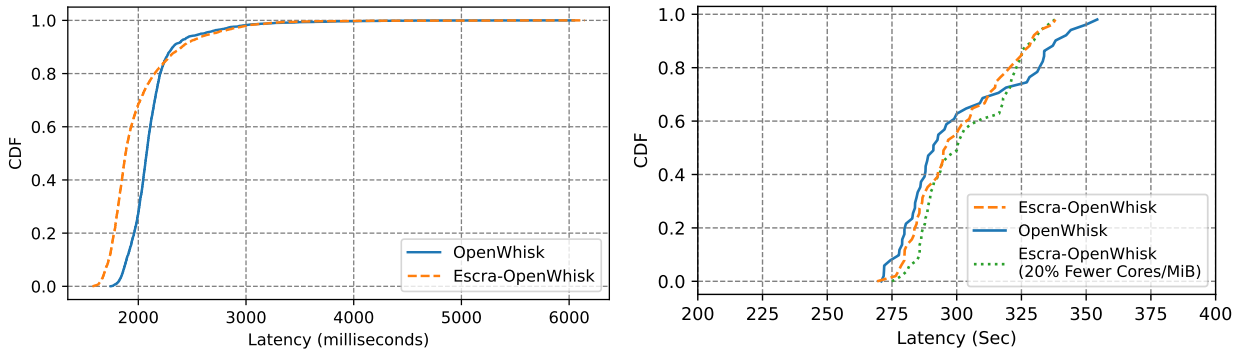
Evaluation Metrics Below are the metrics used in the evaluation of the serverless benchmarks:

- **Aggregate Limits**: Since it is common in serverless systems to bill based on total usage, and serverless providers have a strong incentive to pack as many functions as possible per server, instead of CPU/memory usage per pod we focus on the aggregate of container CPU and memory limits.
- **Application Latency**: Measured in end-to-end latency per request (ImageProcess) or job (GridSearch)

5.6.7 OpenWhisk vs. Escra + OpenWhisk

Performance We first consider ImageProcess performance for OpenWhisk alone and OpenWhisk + Escra. Figure 5.7a shows that, up to the 80th%ile, OpenWhisk + Escra sees modest performance gains over OpenWhisk alone while the overall 99th%ile latency remains similar for both. The average invocation latency with OpenWhisk + Escra is 1.99 seconds as opposed to 2.12 seconds with OpenWhisk alone. Unlike other applications tested with Escra, ImageProcess requires Escra to handle a variable number of pods as the number of application pods at the start of each benchmark iteration is zero. The similarity in tail latency between OpenWhisk alone and OpenWhisk + Escra indicates that Escra is capable of supporting the dynamic scale-up of application pods needed in serverless environments.

To obtain a CDF of GridSearch application latency, we ran GridSearch on: (1) OpenWhisk alone, (2) OpenWhisk + Escra with the same amount of resources allocated as in the OpenWhisk alone experiment, and (3) OpenWhisk + Escra with 80% of the application resource limits al-



(a) ImageProcess request latency

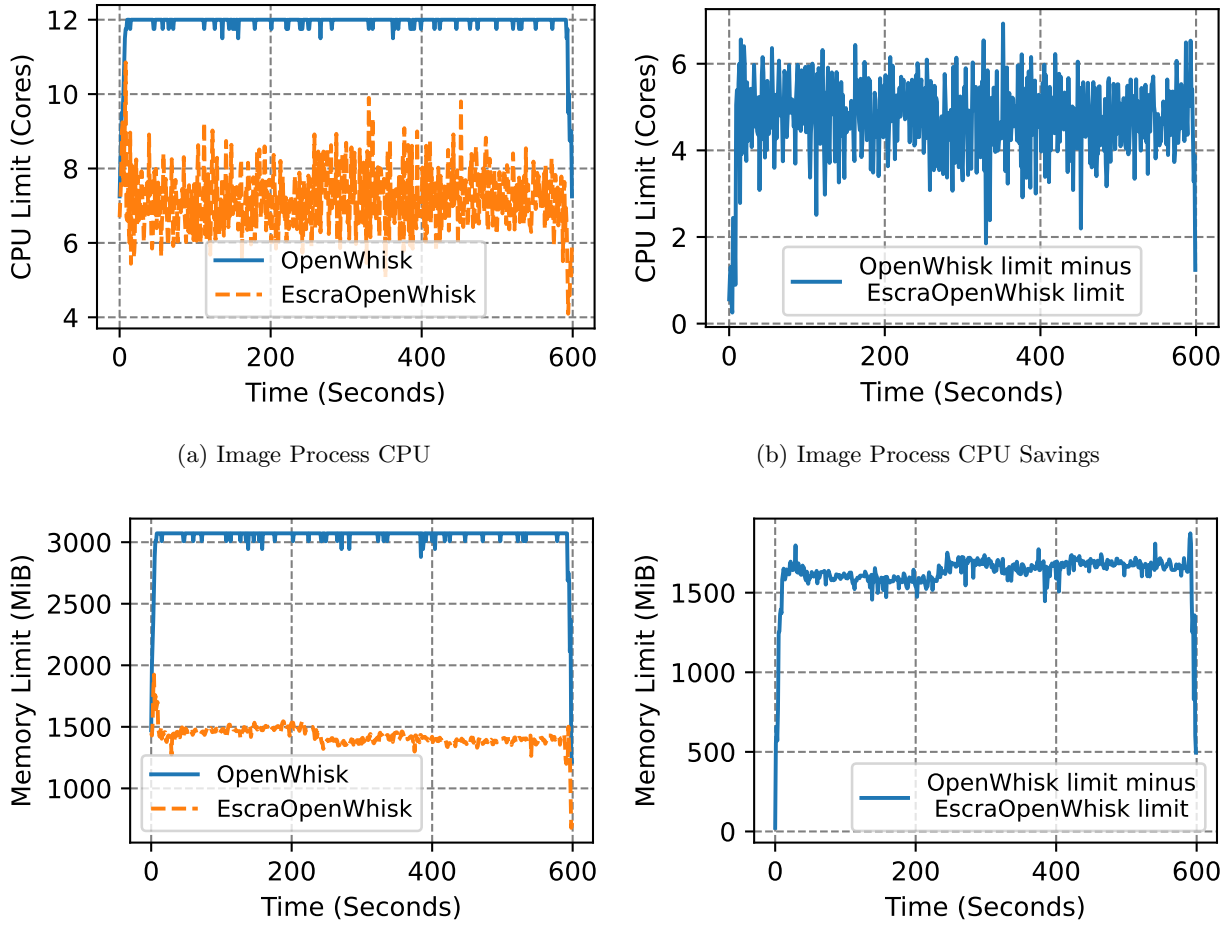
(b) GridSearch application latency

Figure 5.7: Serverless latency CDFs

located compared to OpenWhisk alone. We ran the application 50 times for each configuration. Interestingly, we observe the same average latency (∼300 seconds) when we run GridSearch by allocating equal resources to OpenWhisk and Escra + OpenWhisk (cases 1 and 2) and only 1% higher average (303 seconds) for case 3, showing Escra can allocate fewer resources to an app and maintain similar performance. As is indicated in Figure 5.7b, Escra + OpenWhisk outperforms OpenWhisk alone at 99%ile and has lower tail latency.

Efficiency Figure 5.8 shows aggregate CPU and memory limits for OpenWhisk and OpenWhisk + Escra for ImageProcess. On average, OpenWhisk + Escra sets the limit at 7 vCPU whereas OpenWhisk static allocation results in a limit of 12 vCPU, resulting in a savings of approximately 5 vCPU for identical workloads. For memory, the difference in the limit averages around 1550 MiB.

According to Figure 5.9, OpenWhisk allocates 113 vCPUs for GridSearch on average. On the other hand, Escra + OpenWhisk was able to reduce the vCPU allocation to 53 vCPUs. For memory, on average, OpenWhisk sets the application aggregate limit to 29087 MiB while Escra + OpenWhisk is able to run the same GridSearch application with an application limit of 22264 MiB. On average, Escra + OpenWhisk saves 60 vCPUs and roughly 7 GiB of memory space.



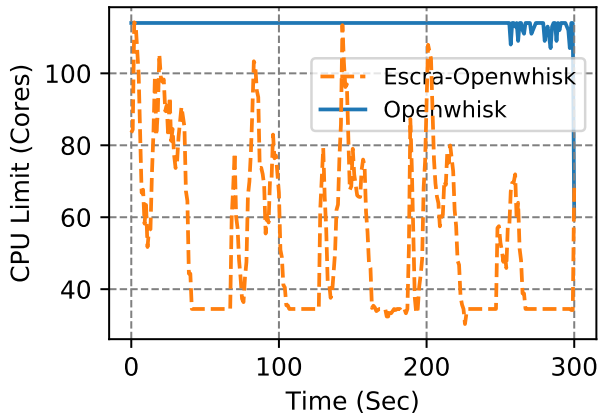
(a) Image Process CPU

(b) Image Process CPU Savings

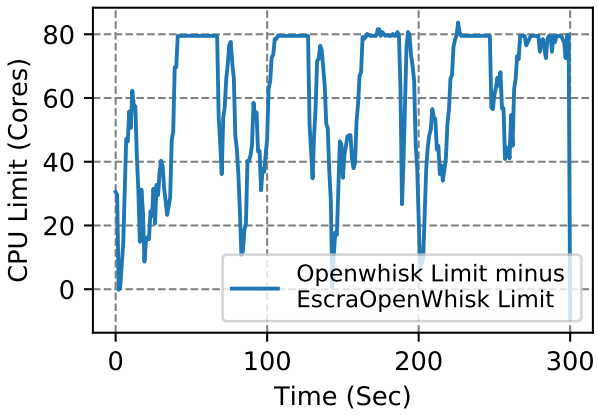
(c) Image Process Mem

(d) Image Process Mem Savings

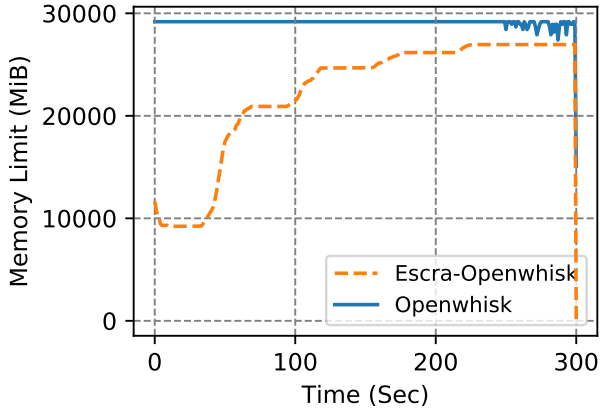
Figure 5.8: Aggregate memory and CPU limits averaged per second over four test iterations for ImageProcess. We highlight the difference (savings) between OpenWhisk limits and OpenWhisk + Escra limits with the savings graphs.



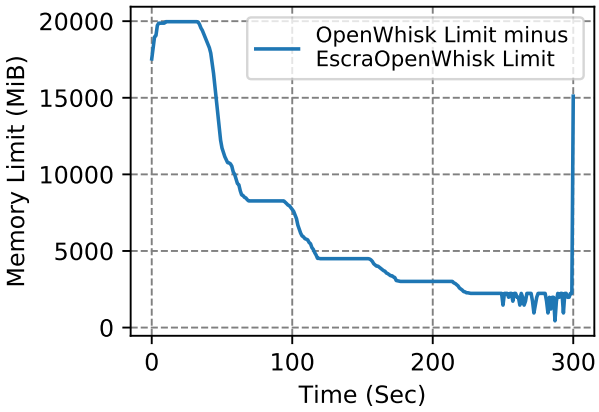
(a) GridSearch CPU



(b) GridSearch CPU Savings



(c) GridSearch Mem



(d) GridSearch Mem Savings

Figure 5.9: Aggregate memory and CPU limits over 5 minutes of running GridSearch. We highlight the difference (savings) between OpenWhisk limits and OpenWhisk + Escra limits with the savings graphs.

5.6.8 Takeaways

As shown in the ImageProcess and GridSearch benchmarks, Escra only minimally effects function latency while providing significant resource savings on static CPU/memory limits. In sum, Escra increased efficiency while maintaining performance. ImageProcess in particular shows that Escra is able to handle a dynamic and rapid increase in number of application pods. The GridSearch results showcases how Escra can help running batch-like, data intensive, long-running applications with fewer resources but without increasing latency.

5.6.9 Escra MicroBenchmarks and Overheads

Why a 100ms Report Period? Escra uses a 100ms CPU telemetry report frequency for two main reasons. First, 100ms complements the default Linux CFS period. Second, we measured the 99% end-to-end latency performance across various report frequencies every 50ms from 50ms to 200ms. Collecting CPU statistics at the end of every period (100ms) and reporting them directly to the controller resulted in the lowest application latency.

Escra Network Overhead Escra sends usage statistics over UDP to the Controller and the Controller launches RPC calls to the Agent process to update container limits. The peak network overhead measured for 32 containers is 12.06 Mbps. Since the majority of the bandwidth usage comes from the per-container CPU telemetry, we expect the network overhead to scale linearly with the number of containers managed. An investigation into how Escra scales as containers are geographically farther away from the Controller and Resource Allocator (increasing network latency) is left to future work.

Escra CPU Overhead The largest CPU consumers in Escra are the Controller, Resource Allocator, and the kernel threads running on each worker node reporting telemetry data. The Controller consumes the most CPU out of the three since the the memory reclamation process relies on the cAdvisor API [28], consuming up to 85% of a core. Replacing the cAdvisor functionality with memory limit/usage system calls would greatly reduce the memory reclamation overhead. Without

cAdvisor, the Controller and Resource Allocator together use 5.7% of a core with 68 containers. For a cloud-scale analysis, we assume a separate Escra Controller and Resource Allocator that manage each application. Escra Controllers and Allocators are able to manage 1,192 containers per core. Assuming 20 cores per node, a collection of Escra Controllers and Allocators can manage up to 23,859 containers per node. Note, as more containers are registered with the Controller, the mean time between subsequent container stats increases sublinearly.

5.7 Discussion and Future Work

This section discusses how Escra affects cloud ecosystems and describes some directions for future work.

Multi-tenant Building a fully-fledged cluster management system that takes advantage of Escra remains future work. The contribution of this chapter is that fine-grained, event-driven resource allocation is possible and performs well. While Escra can effectively reduce slack and increase performance, it remains an open question in how such benefits translate to a large-scale, complex, multi-tenant system.

Serverless Our initial implementation of OpenWhisk + Escra is naive in several ways: (1) all containers are treated as the same application; the framework would need to modify this to deploy pods in per-tenant namespaces, and (2) the OpenWhisk invoker remains unaware of the actual CPU and Memory limits being used; it would need to be modified to ingest current usage and limits from Escra. We leave these to future work.

Billing and Accounting Beyond the efficiency benefits of using Escra in serverless systems, the Distributed Container abstraction may further be useful for billing and accounting in serverless systems [142, 333]. Many commercial frameworks set global limits on serverless applications by setting an invocation limit (i.e., the maximum number of concurrently running functions). With the Distributed Container abstraction, it would be possible to instead limit based on maximum memory or CPU usage. The study of limits and billing using Distributed Containers in serverless

systems is a subject of future work.

Integrated Scheduler Fine-grained container scaling begs the question: do we need fine-grained container scheduling? Escra looks to improve bin-packing capability on a sub-second time-scale by reducing per-container slack. Would a fine-grained scheduler, that quickly schedules and deploys containers, be able to take advantage of periods with lower utilization and bin-pack containers more efficiently? If we could integrate Escra with a scheduler, we could improve bin packing by predicting future container usage on a node and scheduling new deployments if the node is predicted to be underutilized in the future.

This chapter illustrates how current orchestration systems fail to achieve both high performance and cost efficient container deployments, typically trading performance (throughput, latency) for cost-efficiency (slack) or vice versa. We motivate the need for a fine-grained and seamless container scaling orchestrator and propose a solution: Escra. Escra uses kernel hooks to generate both fine-grained telemetry and OOM handling events that allow a logically-centralized Escra Controller to allocate resources within 100s of milliseconds. Escra's scaling algorithm can be customized based on a developer's application requirements, putting the developer in control over the optimization of their container's scaling decisions. With our scaling algorithm, Escra minimizes CPU slack by over 10x compared to our implementation of Autopilot. Escra also reduces application limits in serverless frameworks, saving more than 2x the CPU and memory resources over a standard serverless deployment. Escra's comparison to static approaches, Autopilot, and OpenWhisk deployments indicates fine-grained container scaling finds the balance between performance and efficiency while maintaining isolation. Escra is open-sourced at `https://github.com/gregcusack/Escra`.

5.8 Conclusion

The demand for cloud computing capacity is continuing to rise. As the number of businesses migrating to the cloud increases, the need for application specific security, performance, and efficiency optimizations is greater than ever. Developers need access to application specific security tools, efficient and scalable network monitoring systems, and dynamic, fine-grained CPU

and memory allocation schemes. Unfortunately, the tools exposed to developers by cloud providers to manage and control an application are rigid and inflexible. This cloud level rigidity prevents developers from optimizing their applications' security, performance, and efficiency. As a result, we set out with the following goal: create and then expose a flexible underlying platform to developers that allows them to optimize their specific applications' security, performance, and efficiency. Fine grained control enables developers to control the underlying compute systems themselves, so that they can build systems that benefit their specific applications.

In this dissertation, we identified the shortcomings and rigidity of the underlying hardware and software systems that control and manage secure hardware, network monitoring, and compute resources. We then built new, programmable platforms that enabled developers to design and build application-specific secure hardware features, network monitoring applications, and compute resource allocation mechanisms. Application specific tools that enable fine-grained control over the underlying compute infrastructure will continue to be a necessity as developers look to squeeze the strongest security and highest performance and efficiency out of their applications.

Bibliography

[1] http://programmablelogicinpractice.com/?p=87.

[2] http://www.opensource.apple.com/source/CommonCrypto/CommonCrypto-55010/Source/libtomcrypt/src/ciphers/ltc_aes/aes.c.

[3] http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf.

[4] 128bit hash comparison with sse.

[5] 40Gbit AES Encryption Using OpenCL and FPGAs. http://www.nallatech.com/40gbit-aes-encryption-using-opencl-and-fpgas.

[6] Achieve power-efficient acceleration with opencl on altera fpgas. http://www.altera.com/products/software/opencl/opencl-index.html.

[7] Alibaba cluster trace program. https://github.com/alibaba/clusterdata.

[8] Altera socs. https://www.altera.com/products/soc/overview.html.

[9] Amazon EC2 F1 Instances: Run Customizable FPGAs in the AWS Cloud. https://aws.amazon.com/ec2/instance-types/f1/.

[10] Amazon elastic container service. https://aws.amazon.com/ecs/?whats-new-cards.sort-by=item.additionalFields.postDateTime&whats-new-cards.sort-order=desc.

[11] Amazon elastic kubernetes service (eks). https://aws.amazon.com/eks/.

[12] Android 4.2.2 on zynq getting started guide. http://www.wiki.xilinx.com/Android+4.2.2+On+Zynq+Getting+Started+Guide.

[13] Android native development kit. https://developer.android.com/tools/sdk/ndk/index.html.

[14] Android on zynq getting started guide. http://www.wiki.xilinx.com/Android+On+Zynq+Getting+Started+Guide.

[15] Android software development kit. https://developer.android.com/sdk/index.html.

[16] Apache openwhisk. https://github.com/apache/openwhisk.

[17] Apache storm.

[18] ARM Mali OpenCL SDK. http://malideveloper.arm.com/resources/sdks/mali-opencl-sdk/.

[19] ARM TrustZone. https://www.arm.com/products/security-on-arm/trustzone.

[20] Assign memory resources to containers and pods. https://kubernetes.io/docs/tasks/configure-pod-container/assign-memory-resource/.

[21] Autoscaling. https://docs.aws.amazon.com/eks/latest/userguide/autoscaling.html.

[22] Aws lambda. https://aws.amazon.com/lambda/.

[23] Aws lambda enables functions that can run up to 15 minutes. https://aws.amazon.com/about-aws/whats-new/2018/10/aws-lambda-supports-functions-that-can-run-up-to-15-minutes/.

[24] Aws nitro system. https://aws.amazon.com/ec2/nitro/.

[25] Axi reference guide. http://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf.

[26] Azure functions. https://azure.microsoft.com/en-us/services/functions/.

[27] Azure kubernetes service (aks). https://azure.microsoft.com/en-us/services/kubernetes-service/#overview.

[28] cadvisor. https://github.com/google/cadvisor.

[29] Ces: Intel goes for self-driving cars. https://www.electronicsweekly.com/news/design/ces-intel-goes-self-driving-cars-2017-01/.

[30] Cgroups. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[31] The chromium projecst: Tpm usage.

[32] client-go. https://github.com/kubernetes/client-go.

[33] Cloud functions. https://cloud.google.com/functions.

[34] Cloud functions execution environment. https://cloud.google.com/functions/docs/concepts/exec#timeout.

[35] Cloudwatch metrics for your transit gateways. https://docs.aws.amazon.com/vpc/latest/tgw/transit-gateway-cloudwatch-metrics.html.

[36] Cni - the container network interface. https://github.com/containernetworking/cni.

[37] Data Plane Development Kit.

[38] Device tree. http://elinux.org/Device_Trees.

[39] Dlib c++ library. http://dlib.net.

[40] Docker. https://www.docker.com/.

[41] Docker should assist bandwidth limiting containers. https://github.com/moby/moby/issues/26767.

[42] Enable software programmable digital pre-distortion in cellular radio infrastructure. http://www.techonlineindia.com/techonline/news_and_analysis/169024/enable-software-programmable-digital-pre-distortion-cellular-radio-infrastructure.

[43] Fairphone. https://www.fairphone.com/.

[44] FBI Apple encryption dispute. https://en.wikipedia.org/wiki/FBIApple_encryption_dispute.

[45] Filesystem in userspace. http://fuse.sourceforge.net/.

[46] FPGA Accelerators in GNU Radio with Xilinx's Zynq System on Chip. https://gnuradio.org/redmine/projects/gnuradio/wiki/Zynq/.

[47] FPGA System Smokes Spark on Streaming Analytics. www.datanami.com/2015/03/10/fpga-system-smokes-spark-on-streaming-analytics/.

[48] Gartner forecasts worldwide public cloud revenue to grow 17.5 percent in 2019. https://www.gartner.com/en/newsroom/press-releases/2019-04-02-gartner-forecasts-worldwide-public-cloud-revenue-to-g.

[49] GNU Radio on Android. http://gnuradio.org/redmine/projects/gnuradio/wiki/Android/.

[50] Google Project Ara. http://www.projectara.com/.

[51] GPGPU OpenCL API. http://www.vivantecorp.com/index.php/en/technology/gpgpu.html.

[52] Grafana.

[53] gRPC.

[54] hey. https://github.com/rakyll/hey.

[55] Hipster shop: Cloud-native microservices demo application. https://github.com/Brown-NSG/microservices-demo.

[56] host.json reference for azure functions 2.x and later. https://docs.microsoft.com/en-us/azure/azure-functions/functions-host-json#functiontimeout.

[57] How amazon cloudwatch works. https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/cloudwatch_architecture.html.

[58] Hyperparameter tuning grid search example. https://github.com/lithops-cloud/applications/tree/master/sklearn.

[59] Ibm cloud functions. https://www.ibm.com/cloud/functions.

[60] Imdb. https://www.imdb.com/.

[61] Intel Altera Acquisition. https://newsroom.intel.com/news-releases/intel-completes-acquisition-of-altera/.

[62] Intel OpenCL SDK. https://software.intel.com/en-us/intel-opencl.

[63] Intel Software Guard Extensions. https://software.intel.com/en-us/sgx.

[64] Intel Software Guard Extensions (SGX): A Researcher's Primer. https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2015/january/intel-software-guard-extensions-sgx-a-researchers-primer/.

[65] Intel Trusted Execution Technology: Software Development Guide. https://www-ssl.intel.com/content/www/us/en/software-developers/intel-txt-software-development-guide.html.

[66] Introducing the Intel Software Guard Extensions Tutorial Series. https://software.intel.com/en-us/articles/introducing-the-intel-software-guard-extensions-tutorial-series.

[67] iOS Security - iOS 11. https://www.apple.com/business/docs/iOS_Security_Guide.pdf.

[68] Jetstream network analytics, project website. http://www.jetstream-analytics.net.

[69] Kubelet/kubernetes should work with swap enabled. https://github.com/kubernetes/kubernetes/issues/53533.

[70] Kubernetes cni explained. https://www.tigera.io/learn/guides/kubernetes-networking/kubernetes-cni/.

[71] Kubernetes: Production-grade container orchestration. https://kubernetes.io/.

[72] Kubernetes vertical pod autoscaler. https://github.com/prometheus/prometheus.

[73] LG G5. http://www.lg.com/us/mobile-phones/g5.

[74] libfuse. https://github.com/libfuse/libfuse.

[75] Linux containers (lxc). https://linuxcontainers.org/.

[76] List of data breaches. https://en.wikipedia.org/wiki/List_of_data_breaches.

[77] Lithops. https://lithops-cloud.github.io/.

[78] MicroBlaze Soft Procesor Core. https://www.xilinx.com/products/design-tools/microblaze.html.

[79] Microsemi: Security. https://www.microsemi.com/product-directory/fpga-soc/1738-security.

[80] Nagios monitoring.

[81] One year using kubernetes in production: Lessons learned. https://techbeacon.com/devops/one-year-using-kubernetes-production-lessons-learned.

[82] ONOS.

[83] OpenCL. https://www.khronos.org/opencl/.

[84] Openssl. https://www.openssl.org/.

[85] Openwhisk system details and limits. https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-limits.

[86] Orbot. https://guardianproject.info/apps/orbot.

[87] P4runtime.

[88] Partial reconfiguration user guide. http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_2/ug702.pdf.

[89] Performance co-pilot (pcp) manual. https://pcp.io/docs/index.html.

[90] PowerVR SDK. https://community.imgtec.com/developers/powervr/.

[91] Project catapult. https://www.microsoft.com/en-us/research/project/project-catapult/.

[92] Protocol buffers.

[93] Puzzlephone. http://www.puzzlephone.com/.

[94] Qualcomm Adreno GPU SDK. https://developer.qualcomm.com/software/adreno-gpu-sdk/tools.

[95] Quotas and limit ranges. https://docs.openshift.com/online/pro/dev_guide/compute_resources.html.

[96] Resource management guide - introduction to cgroups. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01.

[97] Resource quotas. https://kubernetes.io/docs/concepts/policy/resource-quotas/.

[98] scikit-learn: Machine learning in python. https://scikit-learn.org/stable/.

[99] Secure Boot Overview. https://technet.microsoft.com/en-us/library/hh824987.aspx?f=255&MSPPError=-2147217396.

[100] Secure Golden Key Boot. https://rol.im/securegoldenkeyboot/.

[101] Sonata source code.

[102] Spark: Inconsistent performance number in scaling number of cores.

[103] Teastore. https://github.com/DescartesResearch/TeaStore.

[104] Telecom at&t's paradise: 75% of telco's mpls tunnel data traffic now under sdn control. https://www.fiercetelecom.com/telecom/at-t-s-paradise-75-telco-s-mpls-tunnel-data-traffic-now-under-sdn-control.

[105] The USRP Hardware Driver Repository. https://github.com/EttusResearch/uhd/.

[106] TimescaleDB.

[107] Tor source code hacking documentation. https://gitweb.torproject.org/tor.git/tree/doc/HACKING.

[108] Trace statistics for CAIDA passive OC48 and OC192 traces – 2015-02-19.

[109] Train ticket: A benchmark microservice system. https://github.com/FudanSELab/train-ticket.

[110] Universal Software Radio Peripheral (USRP) by Ettus Research. http://www.ettus.com/.

[111] Verified Boot. https://source.android.com/security/verifiedboot/.

[112] Vivado design suite. http://www.xilinx.com/products/design-tools/vivado/.

[113] Vivado high-level synthesis. http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/.

[114] Vivado high-level synthesis user guide. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug902-vivado-high-level-synthesis.pdf.

[115] Vivado user guide. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf.

[116] Vmware nsx data center. https://www.vmware.com/products/nsx.html.

[117] What is a virtual private cloud (vpc)? https://www.cloudflare.com/learning/cloud/what-is-a-virtual-private-cloud/.

[118] What is kubernetes. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

[119] Xilinx linux drivers. http://www.wiki.xilinx.com/Linux+Drivers.

[120] Xilinx partial reconfiguration. http://www.xilinx.com/tools/partial-reconfiguration.htm.

[121] Xilinx uio kernel driver. https://github.com/Xilinx/linux-xlnx/tree/master/drivers/uio.

[122] Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit. https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-es2-g.html.

[123] Zed board. http://www.em.avnet.com/en-us/design/drc/Pages/Zedboard.aspx.

[124] Zedboard android. http://elinux.org/Zedboard_Android.

[125] Zedroid - android (5.0 and later) on zedboard. http://www.slideshare.net/noritsuna/zedroid-android-50-and-later-on-zedboard.

[126] Zynq-7000 All Programmable SoC. http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/.

[127] Zynq UltraScale+ MPSoC. http://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html.

[128] Zynq Ultrascale Technical Reference Manual. https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf.

[129] Microsoft Acquires Komoku. http://www.microsoft.com/security/portal/komoku/, 2008.

[130] Nmap. https://nmap.org/, 2009.

[131] Patator. https://github.com/lanjelot/patator, 2011.

[132] Low orbit ion canon. https://github.com/NewEraCracker/LOIC, 2014.

[133] Botnet ares. https://github.com/sweetsoftware/Ares, 2015.

[134] CVE-2016-3287. Available from MITRE, CVE-ID CVE-2016-3287, July 2016.

[135] CVE-2016-3320. Available from MITRE, CVE-ID CVE-2016-3320, August 2016.

[136] Microsoft security bulletin ms16-094 - important, 2016.

[137] Microsoft security bulletin ms16-100 - important, 2016.

[138] iOS Security Guide. https://www.apple.com/business/docs/iOS_Security_Guide.pdf, 2017.

[139] Overview of bitlocker device encryption in windows 10, 2017.

[140] Marcelo Abranches, Sepideh Goodarzy, Maziyar Nazari, Shivakant Mishra, and Eric Keller. Shimmy: Shared memory channels for high performance inter-container communication. In 2nd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 19), 2019.

[141] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In 2018 USENIX Annual Technical Conference (USENIX ATC 18), pages 923–935, Boston, MA, July 2018. USENIX Association.

[142] Zaid Al-Ali, Sepideh Goodarzy, Ethan Hunter, Sangtae Ha, Richard Han, Eric Keller, and Eric Rozner. Making serverless computing more serverless. In 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pages 456–459, 2018.

[143] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI '10), 2010.

[144] M. Al-Qatf, Y. Lasheng, M. Al-Habib, and K. Al-Sabahi. Deep learning approach combining sparse autoencoder with svm for network intrusion detection. IEEE Access, 6:52843–52856, 2018.

[145] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), pages 469–482, Boston, MA, March 2017. USENIX Association.

[146] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In 28th Annual ACM Symposium on Theory of Computing (STOC '96), 1996.

[147] Altera. An 531: Reducing power with hardware accelerators. 2008.

[148] K. Amiri, Y. Sun, P. Murphy, C. Hunter, J. R. Cavallaro, and A. Sabharwal. Warp, a unified wireless network testbed for education and research. In Proceedings of IEEE MSE, 2007.

[149] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy, HASP, volume 13, 2013.

[150] David G. Andersen, Hari Balakrishnan, Nick Feamster, Teemu Koponen, Daekyeong Moon, and Scott Shenker. Accountable Internet Protocol (AIP). In Proc. ACM SIGCOMM, 2008.

[151] Jason H Anderson. A puf design for secure fpga-based embedded systems. In Proceedings of the 2010 Asia and South Pacific Design Automation Conference, pages 1–6. IEEE Press, 2010.

[152] Ross Anderson. Cryptography and competition policy: issues with'trusted computing'. In Proceedings of the twenty-second annual symposium on Principles of distributed computing, pages 3–10. ACM, 2003.

[153] Apache Software Foundation. Flink. `https://flink.apache.org`.

[154] Apache Software Foundation. Kafka. `http://kafka.apache.org`.

[155] Arvind Arasu, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, and Ramarathnam Venkatesan. A secure coprocessor for database applications. In Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on, pages 1–8. IEEE, 2013.

[156] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel O'Keeffe, Mark L Stillwell, et al. Scone: Secure linux containers with intel sgx. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Savannah, GA, 2016.

[157] Anish Athalye, Nicholas Carlini, and David Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In Proceedings of the 35th International Conference on Machine Learning, ICML 2018, July 2018.

[158] P. Athanas, J. Bowen, T. Dunham, C. Patterson, J. Rice, M. Shelburne, J. Suris, M. Bucciero, and J. Graf. Wires on demand: Run-time communication synthesis for reconfigurable computing. In Proc. International Conference on Field Programmable Logic and Applications (FPL), 2007.

[159] T. Auld, A. W. Moore, and S. F. Gull. Bayesian neural networks for internet traffic classification. IEEE Transactions on Neural Networks, 18(1):223–239, Jan 2007.

[160] Stefan Axelsson. The base-rate fallacy and the difficulty of intrusion detection. ACM Trans. Inf. Syst. Secur., 3(3):186–205, August 2000.

[161] Ahmed M Azab, Kirk Swidowski, Jia Ma Bhutkar, Wenbo Shen, Ruowen Wang, and Peng Ning. Skee: A lightweight secure kernel-level execution environment for arm. In Network & Distributed System Security Symposium (NDSS), 2016.

[162] Benoit Badrignans, Reouven Elbaz, and Lionel Torres. Secure fpga configuration architecture preventing system downgrade. In Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on, pages 317–322. IEEE, 2008.

[163] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun Venkataramani. Energy consumption in mobile phones: A measurement study and implications for network applications. In Proc. ACM SIGCOMM Conference on Internet Measurement Conference (IMC), 2009.

[164] Shumeet Baluja and Ian Fischer. Learning to attack: Adversarial transformation networks. In Proceedings of AAAI-2018, 2018.

[165] Mario Barbareschi, Antonino Mazzeo, and Antonino Vespoli. Network traffic analysis using android on a hybrid computing architecture. In Proceedings of the 13th International Conference on Algorithms and Architectures for Parallel Processing - Volume 8286, ICA3PP 2013, pages 141–148, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

[166] Andrew Baumann. Hardware is the new software. In Proc. Workshop on Hot Topics in Operating Systems (HotOS), pages 132–137, 2017.

[167] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI'14, pages 267–283, Berkeley, CA, USA, 2014. USENIX Association.

[168] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. ACM Trans. Comput. Syst., 33(3), Aug 2015.

[169] Jonathan C Beard, Peng Li, and Roger D Chamberlain. Raftlib: A c++ template library for high performance stream parallel processing. The International Journal of High Performance Computing Applications, 31(5):391–404, 2017.

[170] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. In 7th ACM COnference on emerging Networking EXperiments and Technologies, 2011.

[171] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, et al. Onos: towards an open, distributed sdn os. In Proceedings of the third workshop on Hot topics in software defined networking, pages 1–6, 2014.

[172] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. Journal of Cryptographic Engineering, pages 1–13, 2012.

[173] Ketan Bhardwaj, Ming-Wei Shih, Pragya Agarwal, Ada Gavrilovska, Taesoo Kim, and Karsten Schwan. Fast, scalable and secure onloading of edge functions using airbox. In Edge Computing (SEC), IEEE/ACM Symposium on, pages 14–27. IEEE, 2016.

[174] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Srndic, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time. In ECML/PKDD, 2013.

[175] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning attacks against support vector machines. In Proceedings of the 29th International Coference on International Conference on Machine Learning, ICML'12, pages 1467–1474, USA, 2012. Omnipress.

[176] E. Blem, J. Menon, and K. Sankaralingam. Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures. In High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on, pages 1–12, Feb 2013.

[177] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. SIGCOMM Comput. Commun. Rev., 44(3):87–95, July 2014.

[178] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM, SIGCOMM '13, pages 99–110, New York, NY, USA, 2013. ACM.

[179] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In 11th USENIX Workshop on Offensive Technologies (WOOT 17), Vancouver, BC, 2017. USENIX Association.

[180] Eric A Brewer. Kubernetes and the path to cloud native. In Proceedings of the Sixth ACM Symposium on Cloud Computing, pages 167–167, 2015.

[181] Neil Brown. Control groups, part 4: On accounting. `https://lwn.net/Articles/606004/`.

[182] Sven Bugiel, Stefan Nürnberger, Thomas Pöppelmann, Ahmad-Reza Sadeghi, and Thomas Schneider. Amazonia: When elasticity snaps back. In Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS '11, pages 389–400, New York, NY, USA, 2011. ACM.

[183] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. Queue, 14(1):70–93, 2016.

[184] Krzysztof Cabaj, Marcin Gregorczyk, and Wojciech Mazurczyk. Software-defined networking-based crypto ransomware detection using HTTP traffic characteristics. CoRR, abs/1611.08294, 2016.

[185] Krzysztof Cabaj and Wojciech Mazurczyk. Using software-defined networking for ransomware mitigation: the case of cryptowall. CoRR, abs/1608.06673, 2016.

[186] Blake Caldwell, Sepideh Goodarzy, Sangtae Ha, Richard Han, Eric Keller, Eric Rozner, and Youngbin Im. Fluidmem: Full, flexible, and fast memory disaggregation for the cloud. In 2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS), pages 665–677, 2020.

[187] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In 2017 IEEE Symposium on Security and Privacy, pages 39–57, 2017.

[188] Nicholas Carlini and David Wagner. Audio adversarial examples: Targeted attacks on speech-to-text. In Deep Learning and Security Workshop, 2018.

[189] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. ACM SIGCOMM computer communication review, 37(4):1–12, 2007.

[190] A. M. Caulfield et al. A cloud-scale acceleration architecture. In IEEE/ACM International Symposium on Microarchitecture (MICRO), Oct 2016.

[191] Raghavendra Chalapathy and Sanjay Chawla. Deep learning for anomaly detection: A survey. CoRR, abs/1901.03407, 2019.

[192] Chia-Chen Chang, Shun-Ren Yang, En-Hau Yeh, Phone Lin, and Jeu-Yih Jeng. A kubernetes-based monitoring platform for dynamic cloud resource provisioning. In GLOBECOM 2017-2017 IEEE Global Communications Conference, pages 1–6. IEEE, 2017.

[193] Shuai Che, Jie Li, Jeremy W. Sheaffer, Kevin Skadron, and John Lach. Accelerating compute-intensive applications with gpus and fpgas. In Symposium on Application Specific Processors (SASP), 2008.

[194] Wei Chen, Aidi Pi, Shaoqi Wang, and Xiaobo Zhou. Pufferfish: Container-driven elastic memory management for data-intensive applications. In Proceedings of the ACM Symposium on Cloud Computing, pages 259–271, 2019.

[195] Siddhartha Chhabra, Brian Rogers, Yan Solihin, and Milos Prvulovic. Secureme: a hardware-software approach to full system security. In Proceedings of the international conference on Supercomputing, pages 108–119. ACM, 2011.

[196] Pawel Chodowiec and Kris Gaj. Implementation of the twofish cipher using FPGA devices. Technical report, Electrical and Computer Engineering, George Mason University, 1999.

[197] Benoit Claise, Brian Trammell, and Paul Aitken. RFC 7011: Specification of the ip flow information export (ipfix) protocol for the exchange of flow information. `https://tools.ietf.org/html/rfc7011`, 2013.

[198] Cloudflare. Slowloris. `https://www.cloudflare.com/learning/ddos/ddos-attack-tools/slowloris/`.

[199] C. Conger, R. Hymel, M. Rewak, A. George, and H. Lam. Fpga design framework for dynamic partial reconfiguration. In Proceedings of Reconfigurable Architectures Workshop (RAW), 2008.

[200] Victor Costan and Srinivas Devadas. Intel sgx explained. Technical report, Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint. iacr. org/2016/086.

[201] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. Sanctum: Minimal risc extensions for isolated execution. IACR Cryptology ePrint Archive, 2015:564, 2015.

[202] Aimee Coughlin, Greg Cusack, Jack Wampler, Eric Keller, and Eric Wustrow. Breaking the trust dependence on third party processes for reconfigurable secure hardware. FPGA '19, page 282–291, New York, NY, USA, 2019. Association for Computing Machinery.

[203] Aimee Coughlin, Ali Ismail, and Eric Keller. Apps with hardware: enabling run-time architectural customization in smart phones. In 2016 USENIX Annual Technical Conference (USENIX ATC 16), pages 621–634. USENIX Association, 2016.

[204] Aimee Coughlin, Ali Ismail, and Eric Keller. Apps with hardware: Enabling run-time architectural customization in smart phones. In USENIX Annual Technical Conference (ATC), Denver, CO, 2016.

[205] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: A Stream Database for Network Applications. In Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, SIGMOD '03, pages 647–651, New York, NY, USA, 2003. Association for Computing Machinery.

[206] Manuel Crotti, Maurizio Dusi, Francesco Gringoli, and Luca Salgarelli. Traffic classification through simple statistical fingerprinting. SIGCOMM Comput. Commun. Rev., 37(1):5–16, January 2007.

[207] C. Cullinan, C. Wayant, T. Frattesi, and X. Huang. Computing performance benchmarks among cpu, gpu, and fpga. MathWorks. 2013.

[208] Greg Cusack, Oliver Michel, and Eric Keller. Machine learning-based detection of ransomware using sdn. In Proceedings of the 2018 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization, SDN-NFV Sec'18, page 1–6, New York, NY, USA, 2018. Association for Computing Machinery.

[209] Greg Cusack, Oliver Michel, and Eric Keller. Machine learning-based fingerprinting of network traffic using programmable forwarding engines. NDSS Posters, 2018.

[210] Greg Cusack, Maziyar Nazari, Sepideh Goodarzy, Erika Hunhoff, Prerit Oberai, Eric Keller, Eric Rozner, and Richard Han. Escra: Event-driven, sub-second container resource allocation. In 2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS), pages 313–324, 2022.

[211] Greg Cusack, Maziyar Nazari, Sepideh Goodarzy, Prerit Oberai, Eric Rozner, Eric Keller, and Richard Han. Efficient microservices with elastic containers. In Proceedings of the 15th International Conference on Emerging Networking EXperiments and Technologies, CoNEXT '19 Companion, page 65–67, New York, NY, USA, 2019. Association for Computing Machinery.

[212] Andreas Dandalis and Viktor K. Prasanna. An adaptive cryptographic engine for internet protocol security architectures. ACM Trans. Des. Autom. Electron. Syst., 9(3):333–353, July 2004.

[213] Andreas Dandalis, Viktor K. Prasanna, and Jose D.P. Rolim. A Comparative Study of Performance of AES Final Candidates Using FPGAs. In Cryptographic Hardware and Embedded Systems (CHES), 2000.

[214] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. SIGPLAN Not., 49(4):127–144, February 2014.

[215] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In Proc. USENIX Security Symposium, 2004.

[216] Colin Dixon, Arvind Krishnamurthy, and Thomas E Anderson. An end to the middle. In HotOS, volume 9, pages 2–2, 2009.

[217] D.Koch, C. Beckhoff, and J Teich. Recobus-builder a novel tool and technique to build statically and dynamically reconfigurable systems for fpgas. In Proc. Field Programmable Logic and Applications (FPL), 2008.

[218] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, pages 15–28. ACM, 2009.

[219] Jeff Donahue, Philipp Krähenbühl, and Trevor Darrell. Adversarial feature learning. In International Conference on Learning Representations, 2017.

[220] Nick Duffield, Carsten Lund, and Mikkel Thorup. Estimating flow distributions from sampled flow statistics. In 2003 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '03), 2003.

[221] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In Proceedings of the USENIX Annual Technical Conference (ATC), pages 1–14, July 2019.

[222] Thomas Eisenbarth, Tim Güneysu, Christof Paar, Ahmad-Reza Sadeghi, Dries Schellekens, and Marko Wolf. Reconfigurable trusted computing in hardware. In Proceedings of the 2007 ACM workshop on Scalable trusted computing, pages 15–20. ACM, 2007.

[223] A. J. Elbirt and C. Paar. An FPGA Implementation and Performance Evaluation of the Serpent Block Cipher. In Proc ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA), 2000.

[224] A. J. Elbirt, W. Yip, B. Chetwynd, and C. Paar. An fpga-based performance evaluation of the aes block cipher candidate algorithm finalists. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 9(4):545–557, Aug 2001.

[225] Cristian Estan and George Varghese. New directions in traffic measurement and accounting. SIGCOMM Comput. Commun. Rev., 32(4), 2002.

[226] Europol. Internet organised crime assessment 2016 iocta, 2016.

[227] Europol. Internet organised crime assessment 2017 iocta, 2017.

[228] Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry Ponomarev, Nael Abu Ghazaleh, and Ryan Riley. Iso-x: A flexible architecture for hardware-managed isolated execution. In Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, pages 190–202. IEEE Computer Society, 2014.

[229] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. Robust physical-world attacks on deep learning visual classification. In Computer Vision and Pattern Recognition. IEEE, 2018.

[230] Facebook. Fbflow dataset. `https://www.facebook.com/network-analytics`.

[231] Seyed Kaveh Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C. Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, NSDI'14, pages 533–546, Berkeley, CA, USA, 2014. USENIX Association.

[232] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn: an intellectual history of programmable networks. ACM SIGCOMM Computer Communication Review, 44(2):87–98, 2014.

[233] Cheng Feng, Venkata Reddy Palleti, and Aditya Mathurand Deeph Chana. A systematic framework to generate invariants for anomaly detection in industrial control systems. In Network and Distributed System Security Symposium 2019 (NDSS'19), 2019.

[234] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. ACM SIGPLAN Notices, 46(9), 2011.

[235] T. Frangieh, R. Stroop, P. Athanas, and T. Cervero. A modular based assembly framework for autonmous reconfigurable systems. In Reconfigurable Computing: Architectures, Tools and Applications, ser. Lecture Notes in Computer Science, 2012.

[236] Tannous Frangieh, Richard Stroop, Peter Athanas, and Teresa Cervero. A modular-based assembly framework for autonomous reconfigurable systems. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 7199 LNCS:314–319, 2012.

[237] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 3–18, 2019.

[238] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. Opennf: Enabling innovation in network function control. In Proceedings of the 2014 ACM Conference on SIGCOMM, SIGCOMM '14, pages 163–174, New York, NY, USA, 2014. ACM.

[239] Benjamin Glas, Alexander Klimm, Oliver Sander, Klaus Müller-Glaser, and Jürgen Becker. A system architecture for reconfigurable trusted platforms. In Proceedings of the Conference on Design, Automation and Test in Europe, DATE '08, pages 541–544, New York, NY, USA, 2008. ACM.

[240] Guy Gogniat, Tilman Wolf, Wayne Burleson, Jean-Philippe Diguet, Lilian Bossuet, and Romain Vaslin. Reconfigurable hardware for high-security/high-performance embedded systems: the safes perspective. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 16(2):144–155, 2008.

[241] Sepideh Goodarzy, Maziyar Nazari, Richard Han, Eric Keller, and Eric Rozner. Resource management in cloud computing using machine learning: A survey. In 2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA), pages 811–816, 2020.

[242] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In International Conference on Learning Representations, 2015.

[243] Albert Greenberg, Gisli Hjalmtysson, David A Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A clean slate 4d approach to network control and management. ACM SIGCOMM Computer Communication Review, 35(5):41–54, 2005.

[244] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial examples for malware detection. In Simon N. Foley, Dieter Gollmann, and Einar Snekkenes, editors, Computer Security – ESORICS 2017, pages 62–79, Cham, 2017. Springer International Publishing.

[245] Trusted Computing Group. Trusted Platform Module Main Specification (TPM1.0). `http://www.trustedcomputinggroup.org/resources/tpm_main_specification`, March 2011.

[246] Trusted Computing Group. Trusted Platform Module Library Specification (TPM2.0). `http://www.trustedcomputinggroup.org/resources/tpm_library_specification`, March 2013.

[247] Krzysztof Grygiel and Marcis Wielgus. Kubernetes vertical pod autoscaler. `https://github.com/kubernetes/community/blob/master/contributors/design-proposals/autoscaling/vertical-pod-autoscaler.md`.

[248] Guofei Gu, Roberto Perdisci, Junjie Zhang, Wenke Lee, et al. Botminer: Clustering analysis of network traffic for protocol-and structure-independent botnet detection. In USENIX security symposium, volume 5, pages 139–154, 2008.

[249] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces. In 2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQoS), pages 1–10. IEEE, 2019.

[250] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '18, page 357–371, New York, NY, USA, 2018. Association for Computing Machinery.

[251] Prabhat K. Gupta. Xeon+fpga platform for the data center. The Fourth Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL), June 2015.

[252] Arash Habibi Lashkari, Gerard Draper Gil, Mohammad Mamun, and Ali Ghorbani. Characterization of encrypted and vpn traffic using time-related features. 02 2016.

[253] Arash Habibi Lashkari, Gerard Draper Gil, Mohammad Mamun, and Ali Ghorbani. Characterization of tor traffic using time based features. pages 253–262, 01 2017.

[254] Mark Hamilton and William P Marnane. Implementation of a secure tls coprocessor on an fpga. Microprocessors and Microsystems, 40:167–180, 2016.

[255] Ryan Hand, Michael Ton, and Eric Keller. Active Security. In 12th ACM Workshop on Hot Topics in Networks (HotNets-XII), 2013.

[256] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), pages 71–85, Seattle, WA, April 2014. USENIX Association.

[257] Mark Handley, Vern Paxson, and Christian Kreibich. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In USENIX Security Symposium, 2001.

[258] Mohammad Hashemi, Greg Cusack, and Eric Keller. Stochastic substitute training: A gray-box approach to craft adversarial examples against gradient obfuscation defenses. In Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security, AISec '18, pages 25–36, New York, NY, USA, 2018. ACM.

[259] Mohammad J. Hashemi, Greg Cusack, and Eric Keller. Towards evaluation of nidss in adversarial setting. In Proceedings of the 3rd ACM CoNEXT Workshop on Big DAta, Machine Learning and Artificial Intelligence for Data Communication Networks, Big-DAMA '19, page 14–21, New York, NY, USA, 2019. Association for Computing Machinery.

[260] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: a framework for efficient and scalable offloading of control applications. In Proceedings of the first workshop on Hot topics in software defined networks, pages 19–24, 2012.

[261] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In NSDI, volume 11, pages 22–22, 2011.

[262] Hanan Hindy, David Brosset, Ethan Bayne, Amar Seeam, Christos Tachtatzis, Robert C. Atkinson, and Xavier J. A. Bellekens. A taxonomy and survey of intrusion detection system design techniques, network threats and datasets. CoRR, abs/1806.03517, 2018.

[263] A. Hodjat and I. Verbauwhede. High-throughput programmable cryptocoprocessor. IEEE Micro, 24(3):34–45, May 2004.

[264] E. Horta, J. Lockwood, and D. Parlour. Dynamic hardware plugins in an fpga with partial run-time reconfiguration. In Proceedings of the 39th conference on Design automation, June 2002.

[265] E. L. Horta and J. W. Lockwood. Automated method to generate bitstream intellectual property cores for virtex fpgas. In Proc. International Conference on Field Programmable Logic and Applications (FPL), 2004.

[266] Edson L Horta, John W Lockwood, and Saint Louis. PARBIT : A Tool to Transform Bitfiles to Implement Partial Reconfiguration of Field Programmable Gate Arrays (FPGAs). Technical Report WUCS-01-13, Dept. Comput. Sci., Washington Univ., Saint Louis, MO, 2001.

[267] Andrew bunnie Huang and Sean Cross. Novena: A laptop with no secrets, 2015.

[268] Ted Huffmire, Brett Brotherton, Nick Callegari, Jonathan Valamehr, Jeff White, Ryan Kastner, and Tim Sherwood. Designing secure systems on reconfigurable hardware. ACM Transactions on Design Automation of Electronic Systems (TODAES), 13(3):44, 2008.

[269] Ted Huffmire, Brett Brotherton, Gang Wang, Timothy Sherwood, Ryan Kastner, Timothy Levin, Thuy Nguyen, and Cynthia Irvine. Moats and drawbridges: An isolation primitive for reconfigurable hardware based systems. In IEEE Security and Privacy, 2007.

[270] ICS-CERT. Cyber-attack against Ukrainian critical infrastructure. `www.ics-cert.us-cert.gov/alerts/IR-ALERT-H-16-056-01`, 2016.

[271] Intel Intel. and ia-32 architectures software developer's manual. Volume 3A: System Programming Guide, Part, 1(64), 64.

[272] Intel Corporation. Intrinsics guide.

[273] Intel Corporation. Tofino.

[274] William Jackson. Engineer shows how to crack a 'secure' tpm chip, 2010.

[275] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca L. Braynard. Networking named content. In Proc. Conference on Emerging Networking Experiments and Technologies (CoNEXT), 2009.

[276] Matthew Jagielski, Alina Oprea, Battista Biggio, Chang Liu, Cristina Nita-Rotaru, and Bo Li. Manipulating machine learning: Poisoning attacks and countermeasures for regression learning. In 2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA, pages 19–35, 2018.

[277] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: a highly scalable user-level {TCP} stack for multicore systems. In 11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14), pages 489–502, 2014.

[278] Anuj Kalia, Dong Zhou, Michael Kaminsky, and David G. Andersen. Raising the bar for using gpus in software packet processing. In 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), pages 409–423, Oakland, CA, May 2015. USENIX Association.

[279] Richard M. Karp, Scott Shenker, and Christos H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. ACM Trans. Database Syst., 28(1).

[280] E. Keller. Jroute: A run-time routing api for fpga hardware. In IPDPS Workshops, ser. Lecture Notes in Computer Science, volume 1800, 2000.

[281] Srinidhi Kestur, John D Davis, and Oliver Williams. BLAS Comparison on FPGA,CPU and GPU.

[282] Junaid Khalid, Eric Rozner, Wesley Felter, Cong Xu, Karthick Rajamani, Alexandre Ferreira, and Aditya Akella. Iron: Isolating network-based CPU in container environments. In 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), pages 313–328, Renton, WA, April 2018. USENIX Association.

[283] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. Confluo: Distributed Monitoring and Diagnosis Stack for High-speed Networks. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19), 2019.

[284] Changhoon Kim, Anirudh Sivaraman, Naga Katta, et al. In-band network telemetry via programmable dataplanes. In ACM SIGCOMM '15 Demos, 2015.

[285] Myron King, Jamey Hicks, and John Ankcorn. Software-driven hardware development. In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '15, pages 13–22, New York, NY, USA, 2015. ACM.

[286] M. Klein. Power consumption at 40 and 45 nm. Xilinx. 2009.

[287] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. CoRR, abs/1801.01203, 2018.

[288] Oliver Kömmerling and Markus G Kuhn. Design principles for tamper-resistant smartcard processors. Smartcard, 99:9–20, 1999.

[289] Jingfei Kong, Onur Aciicmez, Jean-Pierre Seifert, and Huiyang Zhou. Deconstructing new cache designs for thwarting software cache-based side channel attacks. In Proceedings of the 2nd ACM workshop on Computer security architectures, pages 25–34. ACM, 2008.

[290] Alexey Kopytov. Sysbench. `https://github.com/akopytov/sysbench`.

[291] Balachander Krishnamurthy, Subhabrata Sen, Yin Zhang, and Yan Chen. Sketch-based change detection: Methods, evaluation, and applications. In 3rd ACM Conference on Internet Measurement (IMC '03), 2003.

[292] Ram Shankar Siva Kumar, Andrew Wicker, and Matt Swann. Practical machine learning for cloud intrusion detection: Challenges and the way forward. In Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security, AISec '17, pages 81–90, New York, NY, USA, 2017. ACM.

[293] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world. In International Conference on Learning Representations, 2017.

[294] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. Embark: securely outsourcing middleboxes to the cloud. In 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16), pages 255–273, 2016.

[295] Yann LeCun, Corinna Cortes, and Christopher JC Burges. The mnist database of handwritten digits. 1998.

[296] Qi Li, Xinwen Zhang, Jean-Pierre Seifert, and Hulin Zhong. Secure mobile payment via trusted computing. In Trusted Infrastructure Technologies Conference, 2008. APTC'08. Third Asia-Pacific, pages 98–112. IEEE, 2008.

[297] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16), pages 311–324, Santa Clara, CA, 2016. USENIX Association.

[298] M. Lichman. UCI machine learning repository. `http://archive.ics.uci.edu/ml`, 2013.

[299] David Lie, John Mitchell, Chandramohan A Thekkath, and Mark Horowitz. Specifying and verifying hardware for tamper-resistant software. In Security and Privacy, 2003. Proceedings. 2003 Symposium on, pages 166–177. IEEE, 2003.

[300] Joshua Lind, Ittay Eyal, Florian Kelbert, Oded Naor, Peter R. Pietzuch, and Emin Gün Sirer. Teechain: Scalable blockchain payments using trusted execution environments. CoRR, abs/1707.05454, 2017.

[301] Joshua Lind, Ittay Eyal, Peter Pietzuch, and Emin Gün Sirer. Teechan: Payment channels using trusted execution environments. arXiv preprint arXiv:1612.07766, 2016.

[302] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on, pages 406–418. IEEE, 2016.

[303] Shaoshan Liu, Rn Pittman, and Alessandro Forin. Energy reduction with run-time partial reconfiguration. Fpga, (September), 2010.

[304] Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song. Delving into transferable adversarial examples and black-box attacks. In International Conference on Learning Representations, 2017.

[305] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In Proceedings of the 2016 ACM SIGCOMM Conference, SIGCOMM '16, page 101–114, New York, NY, USA, 2016. Association for Computing Machinery.

[306] Chengzhi Lu, Kejiang Ye, Guoyao Xu, Cheng-Zhong Xu, and Tongxin Bai. Imbalance in the cloud: An analysis on alibaba cluster trace. In 2017 IEEE International Conference on Big Data (Big Data), pages 2884–2892. IEEE, 2017.

[307] Arna Magnúsardóttir. Malware is moving heavily to https, 2017.

[308] M. Majer, J Teich, A. Ahmadinia, and C. Bobda. The erlangen slot machine: A dynamically reconfigurable fpga-based computer. In VLSI Signal Processing Systems, 2007.

[309] Mehrdad Majzoobi, Farinaz Koushanfar, and Srinivas Devadas. Fpga puf using programmable delay lines. In Information Forensics and Security (WIFS), 2010 IEEE International Workshop on, pages 1–6. IEEE, 2010.

[310] R. K. Malaiya, D. Kwon, J. Kim, S. C. Suh, H. Kim, and I. Kim. An empirical evaluation of deep learning for network anomaly detection. In 2018 International Conference on Computing, Networking and Communications (ICNC), pages 893–898, March 2018.

[311] Tarjei Mandt, Mathew Solnik, and David Wang. Demystifying the secure enclave processor. Black Hat USA, 2016.

[312] Moxie Marlinspike. Technology preview: Private contact discovery for signal. `https://signal.org/blog/private-contact-discovery/`, 2017.

[313] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, NSDI'14, pages 459–473, Berkeley, CA, USA, 2014. USENIX Association.

[314] Anthony McGregor, Mark A. Hall, Perry Lorier, and James Brunskill. Flow clustering using machine learning techniques. In PAM, 2004.

[315] J. T. McHenry, P. W. Dowd, F. A. Pellegrino, T. M. Carrozzi, and W. B. Cocks. An FPGA-based coprocessor for ATM firewalls. In Proc IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)), 1997.

[316] Paul McKenney. Memory barriers: a hardware view for software hackers, 2010.

[317] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. SIGCOMM Comput. Commun. Rev., 38(2):69–74, March 2008.

[318] Scott McMillan and Cameron Patterson. Jbits implementations of the advanced encryption standard (rijndael). In International Conference on Field Programmable Logic and Applications, pages 162–171. Springer, 2001.

[319] Larry McVoy and Carl Staelin. Lmbench: Portable tools for performance analysis. In Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference, ATEC '96, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.

[320] Tom Mendelsohn. Secure boot snafu: Microsoft leaks backdoor key, firmware flung wide open. `https://arstechnica.com/information-technology/2016/08/microsoft-secure-boot-firmware-snafu-leaks-golden-key/`.

[321] Nele Mentens, Kazuo Sakiyama, Lejla Batina, Ingrid Verbauwhede, and Bart Preneel. Fpga-oriented secure data path design: implementation of a public key coprocessor. In Field Programmable Logic and Applications, 2006. FPL'06. International Conference on, pages 1–6. IEEE, 2006.

[322] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S McKinley, and Felix Xiaozhu Lin. StreamBox: Modern Stream Processing on a Multicore Machine. In 2017 USENIX Annual Technical Conference (ATC '17), 2017.

[323] O. Michel, J. Sonchack, E. Keller, and Jonathan M. Smith. PIQ: Persistent interactive queries for network security analytics. In ACM International Workshop on Security in Software Defined Networks and Network Function Virtualization (SDN-NFV Sec. '19), 2019.

[324] Oliver Michel, John Sonchack, Greg Cusack, Maziyar Nazari, Eric Keller, and Jonathan M. Smith. Software packet-level network analytics at cloud scale. IEEE Transactions on Network and Service Management, 18(1):597–610, 2021.

[325] Oliver Michel, John Sonchack, Eric Keller, and Jonathan M. Smith. Packet-level analytics in software without compromises. In 10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '18), 2018.

[326] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: An ensemble of autoencoders for online network intrusion detection. In Network and Distributed System Security Symposium (NDSS '18), 2018.

[327] Andrew W. Moore and Denis Zuev. Internet traffic classification using bayesian analysis techniques. In Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '05, pages 50–60, New York, NY, USA, 2005. ACM.

[328] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: a simple and accurate method to fool deep neural networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 2574–2582, 2016.

[329] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The click modular router. ACM Transactions on Computer Systems, 18:263–297, 2000.

[330] Toshihiro Nakae. `https://github.com/tnakae/DAGMM`, 2018.

[331] Jad Naous, Glen Gibb, Sara Bolouki, and Nick McKeown. Netfpga: Reusable router architecture for experimental research. In Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO), 2008.

[332] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '17, page 85–98, New York, NY, USA, 2017. Association for Computing Machinery.

[333] Maziyar Nazari, Sepideh Goodarzy, Eric Keller, Eric Rozner, and Shivakant Mishra. Optimizing and extending serverless platforms: A survey. In 2021 Eighth International Conference on Software Defined Systems (SDS), pages 1–8, 2021.

[334] Stephen Neuendorffer and Chad Epifanio. Generic partially reconfigured processor systems applied to software defined radio. In Proc. of the Software Defined Radio Forum (SDR), 2007.

[335] D. Newman. Aws nitro enclaves: The aws answer for trusted execution environments. `https://futurumresearch.com/research-notes/aws-nitro-enclaves-the-aws-answer-for-trusted-execution-environments/`.

[336] Thuy Nguyen and Grenville Armitage. Synthetic sub-flow pairs for timely and stable ip traffic identification. In Australian Telecommunication Networks and Application Conference 2006, 2006.

[337] Thuy Nguyen and Grenville Armitage. A survey of techniques for internet traffic classification using machine learning. IEEE Communications Surveys & Tutorials, 10(4), 2008.

[338] Ntop. PF_RING.

[339] Jose Nunez-yanez and Arash Beldachi. Run-time power and performance scaling with CPU-FPGA hybrids. pages 55–60, 2014.

[340] Dino Oliva, Rainer Buchty, and Nevin Heintze. Aes and the cryptonite crypto processor. In Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES '03, pages 198–209, 2003.

[341] Amy Ousterhout, Adam Belay, and Irene Zhang. Just in time delivery: Leveraging operating systems knowledge for better datacenter congestion control. In 11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19), 2019.

[342] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads. In 16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19), pages 361–378, 2019.

[343] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S. Chung. Accelerating deep convolutional neural networks using specialized hardware, February 2015.

[344] OWASP. Sql injection. `https://www.owasp.org/index.php/SQL_Injection`.

[345] OWASP. Xss. `https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)`.

[346] Alex Paek and Duncan Mackay. Implementing carrier phase recovery loop using vivado hls. `http://www.xilinx.com/support/documentation/application_notes/XAPP1173-carrier-loop.pdf`.

[347] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, ASIA CCS '17, pages 506–519, New York, NY, USA, 2017. ACM.

[348] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In Security and Privacy (EuroS&P), 2016 IEEE European Symposium on, pages 372–387. IEEE, 2016.

[349] J. Park, H. Tyan, and C. . J. Kuo. Internet traffic classification for scalable qos provision. In 2006 IEEE International Conference on Multimedia and Expo, pages 1221–1224, July 2006.

[350] C. Patterson, P. Athanas, M. Shelburne, J. Bowen, J. Sur´is, T. Dunham, and J. Rice. Slotless module-based reconfiguration of embedded fpgas. In ACM Trans. Embedd. Comput. Syst, October 2006.

[351] David A. Patterson and John L. Hennessy. Computer organization and design. 2009.

[352] Ed Peterson. XAPP 1323: Developing Tamper-Resistant Designs with Zynq UltraScale+ Devices. https://www.xilinx.com/support/documentation/application_notes/xapp1323-zynq-usp-tamper-resistant-designs.pdf, Aug 2018.

[353] Nick L. Petroni, Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In Proc. USENIX Security Symposium, 2004.

[354] P. Phaal, S. Panchen, and N. McKee. Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks. https://www.ietf.org/rfc/rfc3176.txt.

[355] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP '11, pages 85–100, New York, NY, USA, 2011. ACM.

[356] P. Possa, D. Schaillie, and C. Valderrama. Fpga-based hardware acceleration: A cpu/accelerator interface exploration. In IEEE International Conference on Electronics, Circuits and Systems (ICECS), 2011.

[357] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Jim Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In 41st Annual International Symposium on Computer Architecture (ISCA), June 2014.

[358] Andrew Putnam et al. A reconfigurable fabric for accelerating large-scale datacenter services. In Proc. Annual International Symposium on Computer Architecuture (ISCA), 2014.

[359] Yanjun Qi. Random forest for bioinformatics. In Ensemble machine learning, pages 307–323. Springer, 2012.

[360] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An intelligent fine-grained resource management framework for slo-oriented microservices. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pages 805–825. USENIX Association, November 2020.

[361] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon, et al. ftpm: A firmware-based tpm 2.0 implementation. Microsoft Research, 2015.

[362] Anirudh Ramachandran, Srinivasan Seetharaman, Nick Feamster, and Vijay Vazirani. Fast monitoring of traffic subpopulations. In 8th ACM Conference on Internet Measurement (IMC '08), 2008.

[363] Rapid7. Metasploit. https://www.metasploit.com/, 2011.

[364] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. Planck: Millisecond-scale monitoring and control for commodity networks. In Proceedings of the 2014 ACM Conference on SIGCOMM, SIGCOMM '14, page 407–418, New York, NY, USA, 2014. Association for Computing Machinery.

[365] Gourav Rattihalli, Madhusudhan Govindaraju, Hui Lu, and Devesh Tiwari. Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes. In 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), pages 33–40. IEEE, 2019.

[366] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In Proceedings of the Third ACM Symposium on Cloud Computing, pages 1–13, 2012.

[367] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In Proceedings of the third ACM symposium on cloud computing, pages 1–13, 2012.

[368] Barkly Research. Cerber ransomware: Everything you need to know, 2017.

[369] Moshin Riaz and Howard M Heys. The fpga implementation of the rc6 and cast-256 encryption algorithms. In Electrical and Computer Engineering, 1999 IEEE Canadian Conference on, volume 1, pages 367–372. IEEE, 1999.

[370] Teemu Rinta-aho, Mika Karlstedt, and Madhav P. Desai. The click2netfpga toolchain. In Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12), pages 77–88, Boston, MA, 2012. USENIX.

[371] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS '09, pages 199–212, New York, NY, USA, 2009. ACM.

[372] Luigi Rizzo. netmap: A novel framework for fast packet i/o. In 2012 USENIX Annual Technical Conference (ATC '12), 2012.

[373] Martin Roesch. Snort - lightweight intrusion detection for networks. In 13th USENIX Conference on System Administration (LISA '99), 1999.

[374] Benoit Rostykus and Gabriel Hartmann. Predictive cpu isolation of containers at netflix. `https://netflixtechblog.com/predictive-cpu-isolation-of-containers-at-netflix-91f014d856c7`.

[375] Matthew Roughan, Subhabrata Sen, Oliver Spatscheck, and Nick Duffield. Class-of-service mapping for qos: A statistical signature-based approach to ip traffic classification. In Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement, IMC '04, pages 135–148, New York, NY, USA, 2004. ACM.

[376] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. In 2015 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '15), 2015.

[377] Krzysztof Rzadca, Paweł Findeisen, Jacek Świderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Paweł Krzysztof Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. Autopilot: Workload autoscaling at google scale. In Proceedings of the Fifteenth European Conference on Computer Systems, 2020.

[378] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In Security and Privacy (SP), 2015 IEEE Symposium on, pages 38–54. IEEE, 2015.

[379] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. CoRR, abs/1702.08719, 2017.

[380] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In Proceedings of the 8th ACM European Conference on Computer Systems, pages 351–364, 2013.

[381] Jan Seidl. Goldeneye. `https://github.com/jseidl/GoldenEye`.

[382] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. Design and implementation of a consolidated middlebox architecture. In Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI'12, pages 24–24, Berkeley, CA, USA, 2012. USENIX Association.

[383] Vyas Sekar, Sylvia Ratnasamy, Michael K. Reiter, Norbert Egi, and Guangyu Shi. The middlebox manifesto: Enabling innovation in middlebox deployment. In Proceedings of the 10th ACM Workshop on Hot Topics in Networks, HotNets-X, pages 21:1–21:6, New York, NY, USA, 2011. ACM.

[384] Paul Selkirk and Joachim Strömbergson. `https://trac.cryptech.is/browser/core/rng/trng`.

[385] Daniele Sgandurra, Luis Muñoz-González, Rabih Mohsen, and Emil C. Lupu. Automated dynamic analysis of ransomware: Benefits, limitations and use for detection. CoRR, abs/1609.03020, 2016.

[386] S.Guccione, D. Levi, and P. Sundararajan. Jbits: Java-based interface for reconfigurable computing. In Proc. Conf. on Military and Aerospace Application of Programmable Devices and Technology, 1999.

[387] Jay Shah and Dushyant Dubaria. Building modern clouds: using docker, kubernetes & google cloud platform. In 2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC), pages 0184–0189. IEEE, 2019.

[388] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In 2020 USENIX Annual Technical Conference (USENIX ATC 20), pages 205–218. USENIX Association, July 2020.

[389] D. Shan, F. Ren, P. Cheng, R. Shu, and C. Guo. Micro-burst in data centers: Observations, analysis, and mitigations. In 26th IEEE International Conference on Network Protocols (ICNP '18), 2018.

[390] Iman Sharafaldin, Arash Habibi Lashkari, , and Ali A. Ghorbani. Toward generating a new intrusion detection dataset and intrusion traffic characterization. In 4th International Conference on Information Systems Security and Privacy (ICISSP), 2018.

[391] Mahmood Sharif, Sruti Bhagavatula, Lujo Bauer, and Michael K. Reiter. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16, pages 1528–1540, New York, NY, USA, 2016. ACM.

[392] Naveen Kr Sharma, Antoine Kaufmann, Thomas E Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. Evaluating the power of flexible packet processing for network resource allocation. In 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17), 2017.

[393] Prateek Sharma, Ahmed Ali-Eldin, and Prashant Shenoy. Resource deflation: A new approach for transient resource reclamation. In Proceedings of the Fourteenth EuroSys Conference 2019, pages 1–17, 2019.

[394] Sergey Shekyan. Slowhttptest. `https://github.com/shekyan/slowhttptest`.

[395] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: network processing as a cloud service. ACM SIGCOMM Computer Communication Review, 42(4):13–24, 2012.

[396] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In ACM SIGCOMM Computer Communication Review, volume 45, pages 213–226. ACM, 2015.

[397] Ming-Wei Shih, Mohan Kumar, Taesoo Kim, and Ada Gavrilovska. S-nfv: Securing nfv states by using sgx. In Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization, pages 45–48. ACM, 2016.

[398] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In Network and Distributed System Security Symposium (NDSS), February 2017.

[399] Barry Shteiman. Hulk. `https://www.kitploit.com/2014/04/hulk-web-server-dos-tool.html`.

[400] S. Singh and P. James-Roxby. Lava and JBits: From HDL to Bitstream in Seconds. The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'01), 2001.

[401] John Sonchack, Adam J. Aviv, Eric Keller, and Jonathan M. Smith. Turboflow: Information rich flow record generation on commodity switches. In 13th EuroSys Conference (EuroSys '18), 2018.

[402] John Sonchack, Oliver Michel, Adam J Aviv, Eric Keller, and Jonathan M Smith. Scaling hardware accelerated monitoring to concurrent and dynamic queries with *flow. In 2018 USENIX Annual Technical Conference (ATC '18), 2018.

[403] John Sonchack, Jonathan M Smith, Adam J Aviv, and Eric Keller. Enabling practical software-defined networking security applications with ofx. In NDSS, volume 16, pages 1–15, 2016.

[404] R. K. Soni, N. Steiner, and M. French. Open source bitstream generation. In Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2013.

[405] Niranjan Soundararajan. rSmart: The Reconfigurable (Real) Smartphone. Provocative Ideas session of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), March 2013.

[406] Ioannis Sourdis and Dionisios Pnevmatikatos. Fast, large-scale string match for a 10gbps fpga-based network intrusion detection system. In Field Programmable Logic and Application, 2003.

[407] Anna Sperotto, Gregor Schaffrath, Ramin Sadre, Cristian Morariu, Aiko Pras, and Burkhard Stiller. An overview of ip flow-based intrusion detection. IEEE Communications Surveys & Tutorials, 12(3), 2010.

[408] Standard C++ Foundation. C++11 language extensions — general features.

[409] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full sha-1. Technical report, Cryptology ePrint Archive, Report 2017/190, 2017.

[410] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In Proceedings of the 17th annual international conference on Supercomputing, pages 160–171. ACM, 2003.

[411] Suricata. open source ids / ips / nsm engine. `https://suricata-ids.org/`.

[412] Symantec. What you need to know about the wannacry ransomware, 2017.

[413] Symantec. Internet security threat report. `https://www.symantec.com/security-center/threat-report`, 2019.

[414] Synopsis. Heatbleed. `http://heartbleed.com/`.

[415] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. arXiv preprint arXiv:1312.6199, 2013.

[416] T. T. t. Nguyen and G. Armitage. Training on multiple sub-flows to optimise the use of machine learning classifiers in real-world ip networks. In Proceedings. 2006 31st IEEE Conference on Local Computer Networks, pages 369–376, Nov 2006.

[417] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Distributed network monitoring and debugging with switchpointer. In 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18), 2018.

[418] Sandeep Tamrakar et al. Applications of trusted execution environments (tees). 2017.

[419] Gil Tene. wrk2: a http benchmarking tool based mostly on wrk. `https://github.com/giltene/wrk2`.

[420] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: The next generation. In Proceedings of the Fifteenth European Conference on Computer Systems, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[421] S.M. Trimberger and J.J. Moore. Fpga security: Motivations, features, and applications. Proceedings of the IEEE, 102(8):1248–1265, Aug 2014.

[422] Paul Turner, Bharata B Rao, and Nikhil Rao. Cpu bandwidth control for cfs. In Proceedings of the Linux Symposium, pages 245–254, 2010.

[423] Hilary Tuttle. Ransomware attacks pose growing threat. Risk Management, 63(4):4, 2016.

[424] Twitter. The infrastructure behind twitter - scale.

[425] Twitter. Observability at twitter - technical overview.

[426] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and Adaptable Stream Processing at Scale. In Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17, pages 374–389, New York, NY, USA, 2017. ACM.

[427] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In Proceedings of the European Conference on Computer Systems (EuroSys), Bordeaux, France, 2015.

[428] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In 2015 10th Computing Colombian Conference (10CCC), pages 583–590. IEEE, 2015.

[429] Zhenghong Wang and Ruby B Lee. New cache designs for thwarting software cache-based side channel attacks. In ACM SIGARCH Computer Architecture News, volume 35, pages 494–505. ACM, 2007.

[430] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and R"udiger Kapitza. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In European Symposium on Research in Computer Security, pages 440–457. Springer, 2016.

[431] Samuel Weiser and Mario Werner. Sgxio: Generic trusted i/o path for intel sgx. In Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, CODASPY '17, pages 261–268, New York, NY, USA, 2017. ACM.

[432] Wikipedia. List of data breaches. `https://en.wikipedia.org/wiki/List_of_data_breaches`.

[433] Richard Wilkins and Brian Richardson. Uefi secure boot in modern computer security solutions. In UEFI Forum, 2013.

[434] Kyle Wilkinson. XAPP 1267: Using Encryption and Authentication to Secure an UltraScale/UltraScale+ FPGA Bitstream. `https://www.xilinx.com/support/documentation/application_notes/xapp1267-encryp-efuse-program.pdf`, Aug 2018.

[435] Anthony Williams. C++ Concurrency in Action. Manning, 1 edition, 2012.

[436] Nigel Williams, Sebastian Zander, and Grenville Armitage. A preliminary performance comparison of five machine learning algorithms for practical ip traffic flow classification. SIGCOMM Comput. Commun. Rev., 36(5):5–16, October 2006.

[437] Thomas Wollinger, Jorge Guajardo, and Christof Paar. Security on fpgas: State-of-the-art implementations and attacks. ACM Trans. Embed. Comput. Syst., 3(3):534–574, August 2004.

[438] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In Security and Privacy (SP), 2015 IEEE Symposium on, pages 640–656. IEEE, 2015.

[439] Ting-Fang Yen and Michael K Reiter. Traffic aggregation for malware detection. Lecture Notes in Computer Science, 5137:207–227, 2008.

[440] C. Yin, Y. Zhu, S. Liu, J. Fei, and H. Zhang. An enhancing framework for botnet detection using generative adversarial networks. In 2018 International Conference on Artificial Intelligence and Big Data (ICAIBD), pages 228–234, May 2018.

[441] Joel Yliluoma. Bit mathematics cookbook.

[442] Da Yu, Yibo Zhu, Behnaz Arzani, Rodrigo Fonseca, Tianrong Zhang, Karl Deng, and Lihua Yuan. dshark: A general, easy to program and scalable framework for analyzing in-network packet traces. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19), 2019.

[443] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In 10th USENIX Conference on Networked Systems Design and Implementation (NSDI '13), 2013.

[444] Minlan Yu, Lavanya Jose, and Rui Miao. Software Defined Traffic measurement with OpenSketch. In 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), pages 29–42, Lombard, IL, April 2013. USENIX Association.

[445] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In Proceedings of the ACM Symposium on Cloud Computing, SoCC '20. Association for Computing Machinery, 2020.

[446] Yang Yu, Jun Long, and Zhiping Cai. Network intrusion detection through stacking dilated convolutional autoencoders. Security and Communication Networks, 2017, 2017.

[447] Yifei Yuan, Dong Lin, Ankit Mishra, Sajal Marwaha, Rajeev Alur, and Boon Thau Loo. Quantitative network monitoring with netqre. In 2017 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17), 2017.

[448] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In USENIX Symposium on Networked Systems Design and Implementation (NSDI), San Jose, CA, 2012.

[449] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: A unified engine for big data processing. Communications of the ACM, 59(11), 2016.

[450] S. Zander, T. Nguyen, and G. Armitage. Automated traffic classification and application identification using machine learning. In 30th IEEE Conference on Local Computer Networks (LCN'05), 2005.

[451] Houssam Zenati, Chuan Sheng Foo, Bruno Lecouat, Gaurav Manek, and Vijay Ramaseshan Chandrasekhar. Efficient gan-based anomaly detection. CoRR, abs/1802.06222, 2018.

[452] Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A Lee. AWStream: Adaptive Wide-area Streaming Analytics. In Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '18, pages 236–252, New York, NY, USA, 2018. ACM.

[453] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In Proc. ACM SIGSAC Conference on Computer and Communications Security (CCS), 2016.

[454] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. Homealone: Co-residency detection in the cloud via side-channel analysis. In Proceedings of the 2011 IEEE Symposium on Security and Privacy, SP '11, pages 313–328, Washington, DC, USA, 2011. IEEE Computer Society.

[455] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Scalable, high performance ethernet forwarding with cuckooswitch. In Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies, CoNEXT '13, pages 97–108, New York, NY, USA, 2013. ACM.

[456] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. Packet-level telemetry in large datacenter networks. In Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM '15, page 479–491, New York, NY, USA, 2015. Association for Computing Machinery.

[457] Bo Zong, Qi Song, Martin Renqiang Min, Wei Cheng, Cristian Lumezanu, Daeki Cho, and Haifeng Chen. Deep autoencoding gaussian mixture model for unsupervised anomaly detection. In International Conference on Learning Representations, 2018.