Enabling eBPF on Embedded Systems Through Decoupled Verification

Milo Craun
Virginia Tech
Blacksburg, VA, USA
miloc@vt.edu

Adam Oswald
Virginia Tech
Blacksburg, VA, USA
adamoswald@vt.edu

Dan Williams
Virginia Tech
Blacksburg, VA, USA
djwillia@vt.edu

ABSTRACT

eBPF (Extended Berkeley Packet Filter) is a Linux kernel subsystem that aims to allow developers to write safe and efficient kernel extensions by employing an in-kernel verifier and just-in-time compiler (JIT). We find that verification is prohibitively expensive for resource-constrained embedded systems. To solve this we describe a system that allows for verification to occur outside of the embedded kernel and before BPF program load time. The in-kernel verifier and JIT are coupled so they must be decoupled together. A designated verifier kernel accepts a BPF program, then verifies, compiles, and signs a native precompiled executable. The executable can then be loaded onto an embedded device without needing the verifier and JIT on the embedded device. Decoupling verification and JIT from load-time opens the door to much more than running BPF programs on embedded devices. It allows larger and more expressive BPF programs to be verified, provides a way for new approaches to verification to be used without extensive kernel modification and creates the possibility for BPF program verification as a service.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**; **Operating systems**; • **Computer systems organization → Embedded software**.

KEYWORDS

eBPF, verification as a service, kernel extension

ACM Reference Format:
Milo Craun, Adam Oswald, and Dan Williams. 2023. Enabling eBPF on Embedded Systems Through Decoupled Verification. In *1st Workshop on eBPF and Kernel Extensions (eBPF '23), September 10, 2023, New York, NY, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3609021.3609299

1 INTRODUCTION

Extended Berkeley Packet Filters (referred to as BPF) provide a way for users to extend the Linux kernel safely. BPF programs are attached to the Linux kernel such that whenever a certain event occurs, the program is run. Each program must be verified by a static checker before it is loaded into the kernel to ensure kernel safety.

Embedded systems increasingly run Linux kernels and there is a desire to use BPF programs for increased observability and improved performance on these platforms [3, 18]. Despite kernel support for BPF programs, there are challenges to running BPF programs on embedded systems due to their resource constraints. The compilation from high level languages to BPF bytecode requires a toolchain of software that is large and computationally expensive. This has been mitigated by using another machine to compile the program [2]. However verification still remains in the kernel on the embedded device. As we show in section 2, the cost of verifying BPF programs on embedded devices is up to 70x slower than a server, which is prohibitively expensive.

The solution to this cost is to allow verification of BPF programs to happen outside of the embedded device. However, we cannot remove verification without also considering the just-in-time compiler (JIT) because these two mechanisms are coupled inside the kernel. We present a system that allows for BPF programs to be verified and compiled to native code on a dedicated verifier kernel and then loaded into the embedded kernel. We faced three difficult technical challenges:

- The verifier kernel needs to produce exactly what would have been produced by the embedded kernel;
- The embedded kernel cannot trust that an executable not created by it is safe; and
- The external kernel puts symbols and addresses into the jitted BPF program that are completely foreign to the embedded kernel.

We overcome these challenges by using a virtual machine to verify and JIT the program. The external kernel then combines the jitted native code with metadata outlining the structure of the program as well as the symbols and addresses used to allow the embedded kernel to relocate symbols and addresses. Finally the external kernel signs this combination to show that the program is safe.

Decoupling verification and JIT has much broader implications. It allows for significantly more complex and expressive BPF programs by allowing dedicated systems to spend as much time as needed to verify large programs. It also allows for new approaches to verification and JIT compilation to be more easily used. Additionally, it opens the door for systems like verification as a service and parallel verification. No longer will projects be forced to deeply modify Linux kernel code in order to bring their new BPF ideas into use.

2 BPF ON EMBEDDED SYSTEMS

BPF is a Linux kernel subsystem that allows users to extend the kernel safely. BPF programs are typically written in a high level programming language and then compiled to BPF bytecode. The bytecode is then sent to the in-kernel BPF verifier to statically check the program. To keep verification tractable and timely, the verifier also has strict limitations on the number of instructions that a BPF program can have, and that there can be no unbounded loops [21]. After the bytecode is deemed safe, it is JIT compiled into a native executable. All of this processing occurs when the program is initially loaded into the kernel.

Many embedded systems run Linux kernels instead of custom operating systems. In the hobbyist and maker markets single board computers like the various flavors of Raspberry Pi [29] and BeagleBone [7] run Linux based operating systems. Other projects such as OpenWRT [25], Buildroot [8], and the Yocto Project [28] all seek to make running Linux on embedded devices as simple as possible. In the remainder of this section we provide context on why BPF is useful on embedded systems, as well as highlight two challenges stemming from their resource constraints which are:

(1) Compiling BPF programs requires an external system.
(2) Verification of BPF programs is expensive on embedded systems.

2.1 BPF is Useful on Embedded Systems

Despite not being intended for embedded systems, BPF programs can be quite useful on embedded systems. Currently, BPF is extensively used to achieve system observability [6, 24, 27]. In cluster or server production environments increased observability over system performance is incredibly valuable. For embedded systems it is also valuable, for example in support of a network of internet of things (IoT) sensors or to provide finer grained insight into performance [3, 18].

Another compelling use case is to increase the performance of embedded systems. Multiple projects have used BPF programs to successfully increase performance in systems [1, 10, 32]. In embedded systems, system hardware is even more limited so it is important to maximize the performance of the software that runs on them. Work has been done to use BPF programs on the Raspberry Pi to create a performant data plane for IoT sensors and networks making use of the limited eXpress Data Path (XDP) [1] capabilities of the Pi [18].

BPF programs also present a way to safely and dynamically change the behavior of the kernel without reboot in response to changing conditions. This technique has been explored in server deployments of BPF programs to change the configuration in response to runtime conditions [22]. In the real time operating system community, a work created their own BPF runtime for a non Linux kernel to create a system that allows live patching their embedded devices [13].

These use cases have emerged even with the current difficulties of running BPF programs on these devices. We expect new use cases to be developed when it becomes easier to make use of BPF programs, especially considering the benefits that can be gained from offloading part of or entire workloads into kernel space instead of user space.

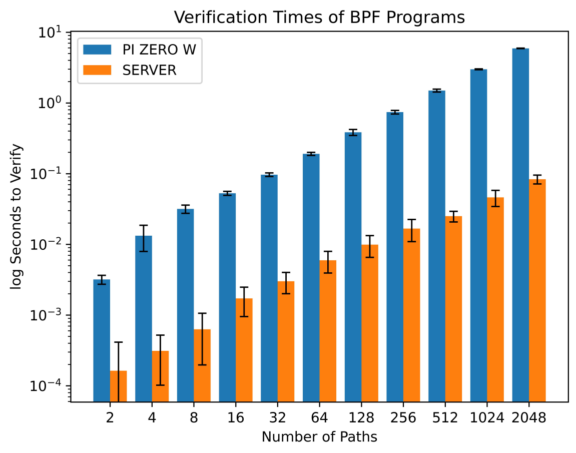

Figure 1: A comparison between the time to verify a BPF program on an embedded system and a more powerful system. Error bars show one standard deviation above and below the mean.

2.2 External BPF Bytecode Compilation

Embedded devices have strict resource constraints in all system resources, including storage. It is imperative to optimally use these resources. Typically BPF programs are compiled to BPF bytecode using LLVM [4]. Projects such as the BPF Compiler Collection (BCC) [15] seek to make building BPF programs easier by providing tools for development and the ability to use Python and Lua as high level languages. The large resource footprint of BPF toolchains makes it hard to justify installing and running them on embedded systems. Compilation is expensive and does not make sense to run on resource constrained systems. Additionally both of these tools are large and place a burden on limited amounts of storage on embedded systems.

These problems have been mostly solved by using an external dedicated machine to compile BPF programs into bytecode. Because BPF bytecode is machine agnostic [11] no cross compiler is needed, and it is easy to compile on an external machine using standard tools.

2.3 Verification is Expensive

A more difficult problem for running BPF programs on embedded devices is that BPF program verification, which occurs every time the program is loaded, is expensive. The verifier uses two passes through the BPF program to verify its safety. The first pass ensures that the program does not exceed the maximum number of allowed instructions defined by the kernel, as well as checking for loops, illegal jumps, and unreachable instructions. The second pass involves walking through all possible program paths to make sure there are no branches where kernel safety is compromised [21]. More complex BPF programs take longer to verify as the possible number of program paths increases significantly.

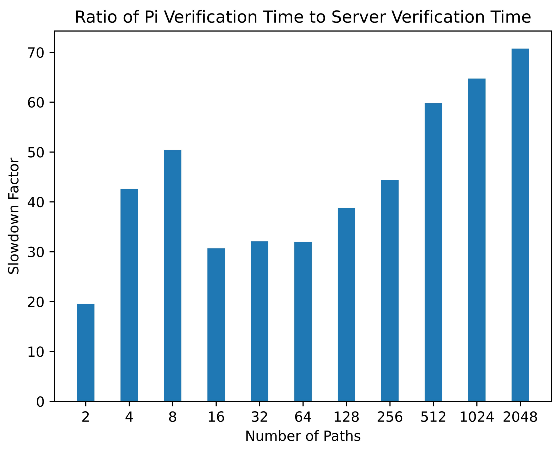

Figure 2: The number of times slower the Pi was to verify the same programs as the server.

For many systems, the cost to verify programs can be minimal, but for less powerful embedded systems, verification can be prohibitively expensive. To quantify this, we conducted an experiment to measure the difference in the time for verification between servers and embedded systems. We started with the downstream Raspberry Pi Linux 6.1.31 kernel, and then added a set of small modifications to measure the time taken to run the verification and JIT functions. We compiled and configured this kernel to run on the Raspberry Pi Zero W and to run in a QEMU virtual machine hosted on a system with an Intel Xeon Gold 5317 processor and 256GB total of system memory. Because we modified the Linux kernel, we decided to use a VM to simplify the process of using the new kernel on our server. We decided to limit the VM to only use 4 CPU cores and 4GB of RAM because currently BPF verification only runs on one CPU core and from informal testing we did not see verification time change based on the available system memory, assuming there was enough available. 4GB of RAM is more than sufficient to verify all of our test programs as well as support the kernel.

We created a series of synthetic BPF programs where we specified the number of possible paths through the program, which is the key factor affecting verification time. We generated programs where the number of paths is a given power of 2. Each program contains nested if statements that take branches based on a random number that is generated after each branch in order to avoid state pruning [21]. Because state pruning can greatly reduce the number of states to check, our results represent more of a worst case than an average case. We used bpftool version 7.1 to load each of our programs into the kernel. The verifier succeeding in verifying all our programs from 2 to 2048 paths before it started rejecting them. With our modified kernel we collected data on the time to verify, JIT, and the total load time for each program. We loaded each program a total of ten times and computed the average of the trials.

Figure 1 and Figure 2 show that there was between a 19x and 70x slowdown in BPF program verification time between the server and the Pi. Additionally, we analyzed the data for the Pi and found that between 91% and 99% of BPF program load time was spent doing verification, showing that JIT compilation does not represent a significant cost. There are many potential confounding factors in the absolute interpretation of the results such as differences in architecture, the overhead of running in a VM, and the differences needed in kernel configuration to run on the Raspberry Pi. Furthermore, we expected there to be a constant factor for slowdown between the Pi and the server, but our data shows that there is another pattern that we cannot currently explain. Regardless, our experiment gives an idea of how much slower embedded systems can be for BPF program verification. From our experiment, we conclude that this degree of slowdown makes it prohibitively expensive to verify BPF programs on embedded hardware, especially as BPF programs evolve to be larger and more complex [10].

3 DECOUPLING VERIFICATION FROM THE KERNEL

We identified five key design goals:

- **Enable pre-verified programs to be loaded into any compatible kernel:** We wanted our solution to be general and not tied to any specific architecture.
- **Provide a way to increase the allowed complexity, and thus expressibility, of BPF programs on embedded systems:** Current embedded systems have limitations on the complexity of BPF programs that they can run because of resource constraints.
- **Allow cross-architecture verification:** We wanted to make use of available performant x86 hardware to run our dedicated verifier.
- **Achieve full compatibility with the Linux verifier:** We wanted our system to be fully equivalent to the current in-kernel verifier except for when verification occurs.
- **Ensure that the safety guarantees made by the verifier still hold for our pre-verified program:** We wanted to make sure that we did not lose any safety properties given by the BPF verifier

To meet our design goals we concluded that the verification of BPF programs should be decoupled from the kernel and moved away from load-time. Because BPF compilation already makes use of an external computer this approach does not add additional hardware requirements. We found that an immediate consequence of this decoupling was that JIT compilation also needed to be moved along with verification. Currently verification and JIT are tightly coupled inside the kernel. For example, the maximum allowed number of tail calls in a BPF program is 33 [5], but we found that this check occurs inside the JIT code, rather than the verifier. The verifier and JIT work together to ensure BPF program safety, so it is nontrivial to separate them. We decided the solution to this is to decouple the JIT along with the verifier. [1] As an added bonus, decoupling the JIT as well increases the flexibility of our system. Users should be able to produce a BPF program, verify and JIT it once per architecture, and then load it into any compatible kernel. This solves the issue of prohibitively expensive verification on embedded machines. A BPF repository of sorts could be created that hosts pre-verified, jitted, and signed BPF programs for anyone to download and load

[1]There are benefits to leaving the JIT in place that we discuss in Section 3.3

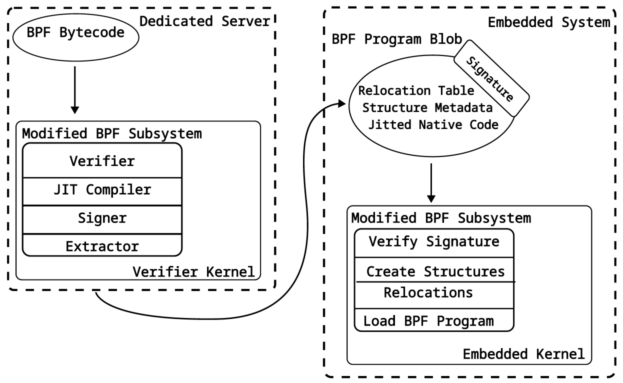

Figure 3: System Design Overview

into their kernel. A further discussion of the benefits to decoupling appears in Section 4.

3.1 Design Challenges

Our design idea and goals led to several main challenges that had to be overcome:

(1) The verifier kernel needs to verify and JIT the program identically to how the embedded kernel would;
(2) The embedded kernel can not trust that the foreign executable created by verifier kernel is safe; and
(3) The jitted native code contains symbols and addresses from the verifier kernel that are totally foreign to the embedded kernel.

3.2 Design Walk-through

Our design in Figure 3 has four main stages. We use a virtual machine running a dedicated verifier kernel which will verify the BPF program. First we load a program in BPF bytecode format into the dedicated verifier kernel. In that kernel we verify the bytecode and then produce a native executable by using the JIT compiler. We have to ensure that the generated native code is for the same architecture as the embedded kernel. In our prototype we achieved this by emulating the architecture of the embedded kernel on an x86_64 server.

Once the program has been verified and jitted, the dedicated kernel collects metadata about the BPF program and combines it with the jitted native code. The dedicated kernel then signs the combined jitted code and metadata. This signature is the guarantee that the program is safe to execute. Because we used the in-kernel verifier, this signature means that the BPF program would have been verified on the embedded device given enough time.

After the program has been signed, we need to extract the combined native code and metadata. Certain parts of the program will need to be relocated when loaded into the embedded kernel. We use part of the metadata to build a relocation table to make relocation easy once the embedded kernel tries to load the program. The rest of the metadata is used to create the necessary kernel structures that would have been created by the verifier.

After extraction we are left with a signed binary blob containing the native executable code for the embedded system, some relocations, and other required metadata. This then gets loaded into the embedded kernel, which makes sure that the signature is valid. If it is invalid, then the program is rejected. Otherwise the embedded kernel goes to work resolving relocations and creating necessary structures. After this is complete we have successfully loaded our BPF program into the embedded kernel.

We will now examine various stages of the design and how they relate to our design goals and challenges.

3.3 Achieving Complete Compatibility

The verifier kernel is a trusted system that handles verifying and jitting the BPF bytecode. Rather than create a custom verifier we make use of the in-place Linux verifier and JIT. These kernel components may contain bugs or quirks that subtly change what programs are allowed and how programs are actually converted to native code. For our goals, we want to ensure that we only load programs that would verify and JIT on the target kernel given enough time and system resources. Our system is then completely compatible with the Linux kernel. As discussed further in Section 4, our system could potentially be expanded to allow other verifiers, like PREVAIL [9], to verify the safety of the BPF bytecode or other JITs to emit customized native code.

Another aspect of compatibility is that our system has to deal with cross-architecture compilation. As mentioned in the walk-through, our initial solution to this is to use a virtual machine emulating the same architecture as the embedded system. This is the most straightforward solution and is viable because the VM only needs to verify and JIT the BPF program one time. Another more complicated solution we are investigating is plugging in the JIT code for other architectures into a kernel native to the dedicated server. For example we could run the ARM JIT on an x86 kernel. This would alleviate the performance overhead of emulating another architecture.

We could avoid challenges with cross-architecture JIT by leaving the JIT in place. This allows for potential device specific JIT optimizations. As shown in Section 2, the time spent on JIT compilation is not a significant contributor to overall BPF program load time. However, to do this we need to break the coupling between the JIT and verifier. We also do not get the benefits of producing precompiled native code. Our initial design is to take the JIT out of the kernel, but we are doing more investigation into which approach is best.

3.4 Ensuring Trust and Safety

The key feature of using BPF is that programs are guaranteed to be safe. By decoupling the verification of BPF programs from the specific kernel that they will be running on, we need a way to communicate that the BPF program is in fact safe. We do this by having the verifying kernel sign the created binary blob. This signature is only as good as the user's trust in how they obtained the binary blob and their trust in the verifying kernel. One use case we envision is individuals using virtual machines to verify and sign their own BPF programs for deployment in their own systems. Safety is then guaranteed. For other applications, the trust needed becomes similar to other Public Key Infrastructure policies. Some central trusted authority would have to create the BPF program blobs, and then distribute them. This approach is well known and straightforward to implement efficiently.

3.5 Relocating Foreign Addresses

Importantly there are two core features of BPF programs that must be relocated when loaded into a new kernel. The first are helper functions which are functions defined in the kernel that BPF programs are able to call. Each helper function has a canonical numbering that is shared between different kernel versions. Due to KASLR, differing kernel configurations, and differing load times and environments, the addresses of helper functions will not be the same between two kernels. To relocate these calls we use two mechanisms. The first is that the verifier kernel creates a relocation table for these functions. The table consists of offsets to call instructions in the jitted native code matched with numbers specifying which helper function is supposed to be called. This information is used by the target kernel when we load the jitted BPF program blob. Once the target kernel has created the necessary kernel structures, we know where the jitted BPF program will be loaded. This allows us to rewrite the binary instructions using the relocation table and kernel functions to resolve symbols to addresses.

The second feature that needs to be relocated are maps. Maps are the primary way for BPF programs to persist data and share it from kernel space into user space. Unlike helper functions, maps are created dynamically and do not have a fixed address in kernel memory. We envision solving map relocations in a similar way to solving helper function relocations. Slightly more work will be needed to locate the address of the needed maps, but once we have that information we should be able to link everything up using our relocation metadata.

3.6 Implementation Status

To date, our implementation effort is at an early stage. As a first step, we are focusing on extracting the jitted BPF program from the verifier kernel and locating the symbols that will need adjustment (e.g., for helper functions).

4 TOWARDS VERIFICATION AS A SERVICE

Although we have motivated decoupling as a way to enable BPF programs to be run on embedded systems, it also opens new and exciting opportunities for the BPF ecosystem:

(1) Decoupling allows increased BPF program complexity;
(2) Decoupling provides a way to expand the verification and JIT ecosystem surrounding BPF; and
(3) Decoupling allows new approaches to be taken to BPF program development and infrastructure.

4.1 Raising BPF Limits

Currently the verifier places strict limits on the number of instructions that a BPF program can have, and the number of instructions that the verifier can check [20]. The intention is to force BPF program verification and JIT to run in a reasonable amount of time. By decoupling verification and JIT from load time we allow an increase to the verifier limits.

From our experiment we showed that there is a large disparity between program verification time on differently sized hardware. We want to use our system to investigate how much more complicated we can make BPF programs when they have large amounts of computing horsepower behind them. Powerful systems can verify large programs much faster than less powerful systems. Decoupling allows these systems to spend as much time as needed to verify BPF programs of any complexity. The output of this is a safe BPF program that does not need to be verified ever again.

Some current projects are already pushing the bounds for what is possible with the verifier. BMC uses BPF programs to significantly increase the performance of Memchached [10]. Their program consisted of 513 lines of C code and required 7 different BPF programs to implement. This technique is called chaining, and incurs overhead. Each program uses a tail call [5] to call into the other parts of the program. BPF programs can also use BPF-to-BPF calls [5] to call into other verified BPF programs. The KFuse [17] system minimizes this overhead for tail calls, but does not mitigate the decreased ability of programmers to reason about their programs. Having to separate programs based on verifier constraints rather than natural program boundaries is not a good solution to building larger BPF programs.

Work on implementing complex network services using BPF has been promising [22]. This work features many tricks to create larger and more complicated BPF programs that can still be verified. One such trick they used was to send packets into user-space and then process them there. They found a significant reduction in performance from this approach. Limitations from the verifier forced them to implement their extensions in a less performant way than the in-kernel network stack. For these kind of extensions to be valuable they need performance at least comparable to in-kernel processing. If the complexity allows it, we could replace entire kernel components with user provided BPF programs.

4.2 Expanding the Ecosystem

Currently BPF verification and JIT exist in an isolated system inside the kernel without a good way to incorporate new tools. Work has been done to create a new verifier, PREVAIL, with a formal basis [9]. The PREVAIL verifier has better asymptotic complexity than the in-kernel verifier, but without additional work, it cannot be used when loading real BPF programs into the Linux kernel. However, PREVAIL is the verifier for eBPF For Windows [30], which shows that the BPF community is interested in PREVAIL and has trust in its ability to verify BPF programs. Work has also been done to formally verify the in-kernel JIT [23], which led to multiple bug fixes and JIT improvements being incorporated into the Linux kernel. Verifier and JIT design incorporate trade-offs, and there may not be a one size fits all solution. Custom components may perform significantly better than the general in-place verifier and JIT. Despite the potential for custom verifiers and JITs, the current design of the BPF subsystem does not allow them without extensive kernel modifications. The problem of verification and JIT is more nuanced than a tightly coupled in-kernel system can allow. Decoupling verification and JIT from the kernel allows the ecosystem to expand by providing an easier way to incorporate new projects and ideas.

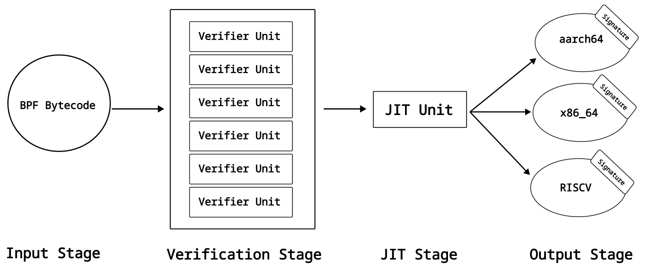

Figure 4: One possible design for distributed verification producing executables for multiple platforms.

4.3 Allow New Approaches

Decoupling verification allows for new approaches to be made for BPF programs. An interesting idea is whether or not BPF program verification can happen with some form of parallelization. If state is copied and information about what branches are to be checked can be communicated and synchronized between processes, then we may be able to further increase the size and complexity of our BPF programs. It also opens the possibility of having distributed BPF program verification, where many machines each verify chunks of a single large BPF program. One initial design idea is shown in Figure 4. Without decoupling verification from load time, these would not be feasible. An offshoot of this is the potential industry of verification as a service. Users upload BPF bytecode and get back a verified BPF binary blob that can then be loaded.

BPF bytecode is machine agnostic, so it should be able to be run on any machine that supports BPF. Efforts such as BTF and Compile Once Run Everywhere (CO-RE) [11] aim to support this on a bytecode level. Notice that a program that is verified once, should always be verified assuming it has not been changed. Can we incorporate these ideas to create a Verify Once Run Everywhere (VO-RE) system for BPF? Implementing this means that one could create a super kernel that can verify, compile, and sign BPF programs for any supported architecture, potentially at the same time. Alternatively we could create BPF fat binaries that contain native instructions for a few common architectures.

5 RELATED WORK

Our work is central to a large collection of additional work. Projects such as PREVAIL [9] aim to create a more efficient BPF verifier that also has a robust formal foundation. Changing the asymptotic bounds of verification allows much more complicated programs to be verified, which increases the expressibility of BPF programs. We also see projects like K2 [31] which aim to generate more optimized but equivalent BPF bytecode.

Recent work has made the claim that kernel verification is untenable [16]. They propose that BPF extensions should be written in Rust in a system that guarantees safety through a combination of runtime mechanisms and language safety features. Their main concerns with the verifier are that it places unreasonable constraints on BPF programs and that it cannot verify the safety of helper functions. Our system works to alleviate the first concern, and creates a platform to provide verification for helper functions.

All of these works seek, at least in part, to make BPF a more expressive programming language. They also present opportunities for our decoupling work. Moving verification and JIT outside the kernel allows other non-kernel programs to handle verification and JIT. This allows for customizable verifiers with more or less properties that they guarantee.

The BPF-for-Windows [30] project uses an external verifier to verify BPF programs. Like our design it incorporates a signature scheme to show to the OS kernel that the program is safe. It uses non-kernel tools to provide verification and JIT capabilities on Windows.

Outside of BPF verification, many projects are focusing on increasing the extensibility of the Linux kernel utilizing BPF programs and traditional kernel extensions. One project seeks to allow users to modify kernel locking behavior [26]. It makes use of BPF and Livepatch to modify the running kernel in place. Other projects seek to include BPF to speed up core operating system tasks. eXpress Data Path (XDP) [1] and eXpress Resubmission Path (XRP) [32] seek to utilize BPF to improve the performance of the networking stack and IO respectively.

We can also look to the Java programming language for inspiration. It implements the Java Virtual Machine, and compiles Java code into Java bytecode [19]. Java bytecode is then JIT compiled again into native machine code. There are some similarities between Java and BPF that may provide insights. Throughout Java's history, there has been work to improve its JIT. One attempt to make the Java JIT better used trace based compilation [12]. Instead of compiling whole functions, the JIT focused on compiling frequently executed paths through the program. Another work focused on tuning the JVM JIT automatically to achieve the best performance [14]. The BPF JIT may be able to benefit from the history of work on the Java JIT, especially if larger and more complicated pieces of software are able to be verified.

6 CONCLUSION

There is a growing need and interest for custom extensions to the Linux kernel, but kernel programming is hard and can have disastrous consequences. BPF programs present a powerful tool to allow users to safely modify and extend the kernel without needing in-depth knowledge of Linux kernel internals. An underappreciated use case for BPF programs is in embedded systems. In this work we presented a system to allow embedded systems to make use of all the great benefits associated with the BPF subsystem. Decoupling verification is also the first step in growing a new BPF ecosystem with more complex programs and more desirable use cases. BPF is a powerful tool for kernel extensions with use cases far beyond what it is used for now. Expanding the range of machines that BPF can run on is one step towards realizing the potential of BPF. This work raises no ethical considerations.

ACKNOWLEDGEMENT

This work is supported in part by NSF grant CNS-2236966.

REFERENCES

[1] Paolo Abeni. 2018. Achieving high-performance, low-latency networking with XDP: Part I. https://developers.redhat.com/blog/2018/12/06/achieving-high-performance-low-latency-networking-with-xdp-part-1

[2] Adrian Ratiu. 2019. An eBPF overview, part 4: Working with embedded systems. https://www.collabora.com/news-and-blog/blog/2019/05/06/an-ebpf-overview-part-4-working-with-embedded-systems/

[3] Adrian Ratiu. 2022. Tracing resource-constrained embedded systems using eBPF. https://elinux.org/images/2/22/Embedded-eBPF.pdf

[4] Alan Maguire. 2019. BPF In Depth: Building BPF Programs. https://blogs.oracle.com/linux/post/bpf-in-depth-building-bpf-programs.

[5] Cilium Authors. 2023. https://docs.cilium.io/en/latest/bpf/architecture/

[6] The Cilium Authors. 2023. Cilium. https://github.com/cilium/cilium

[7] BeagleBoard.org Foundation. 2023. BeagleBone. https://beagleboard.org/bone.

[8] Buildroot. 2023. Buildroot. https://buildroot.org/.

[9] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. 2019. Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 1069–1084. https://doi.org/10.1145/3314221.3314590

[10] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. 2021. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 487–501. https://www.usenix.org/conference/nsdi21/presentation/ghigoff

[11] Brendan Gregg. 2020. https://www.brendangregg.com/blog/2020-11-04/bpf-co-re-btf-libbpf.html

[12] Christian Häubl and Hanspeter Mössenböck. 2011. Trace-Based Compilation for the Java HotSpot Virtual Machine. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ '11)*. Association for Computing Machinery, New York, NY, USA, 129–138. https://doi.org/10.1145/2093157.2093176

[13] Yi He, Zhenhua Zou, Kun Sun, Zhuotao Liu, Ke Xu, Qian Wang, Chao Shen, Zhi Wang, and Qi Li. 2022. RapidPatch: Firmware Hotpatching for Real-Time Embedded Devices. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 2225–2242. https://www.usenix.org/conference/usenixsecurity22/presentation/he-yi

[14] Kenneth Hoste, Andy Georges, and Lieven Eeckhout. 2010. Automated Just-in-Time Compiler Tuning. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '10)*. Association for Computing Machinery, New York, NY, USA, 62–72. https://doi.org/10.1145/1772954.1772965

[15] IO Visor. 2023. BPF Compiler Collection. https://github.com/iovisor/bcc.

[16] Jinghao Jia, Raj Sahu, Adam Oswald, Dan Williams, Michael V. Le, and Tianyin Xu. 2023. Kernel Extension Verification is Untenable. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HOTOS '23)*. Association for Computing Machinery, New York, NY, USA, 150–157. https://doi.org/10.1145/3593856.3595892

[17] Hsuan-Chi Kuo, Kai-Hsun Chen, Yicheng Lu, Dan Williams, Sibin Mohan, and Tianyin Xu. 2022. Verified Programs Can Party: Optimizing Kernel Extensions via Post-Verification Merging. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 283–299. https://doi.org/10.1145/3492321.3519562

[18] Douglas J. Paul Kyle A. Simpson, Chris Williamson and Dimitrios P. Pezaros. 2023. GALETTE: a Lightweight XDP Dataplane on your Raspberry Pi. In *IFIP Networking 2023*. Barcelona, Spain. Accepted for Publication.

[19] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2014. *The Java Virtual Machine Specification, Java SE 8 Edition* (1st ed.). Addison-Wesley Professional.

[20] Linux. 2023. BPF Design Q&A. https://docs.kernel.org/bpf/bpf_design_QA.html. Accessed: 2023-06-06.

[21] Linux. 2023. eBPF verifier. https://docs.kernel.org/bpf/verifier.html. Accessed: 2023-06-05.

[22] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. 2018. Creating Complex Network Services with eBPF: Experience and Lessons Learned. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. 1–8. https://doi.org/10.1109/HPSR.2018.8850758

[23] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. 2020. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 41–61. https://www.usenix.org/conference/osdi20/presentation/nelson

[24] NetObserv. 2023. NetObserv. https://github.com/netobserv/network-observability-operator

[25] OpenWrt. 2023. OpenWrt. https://openwrt.org/.

[26] Sujin Park, Diyu Zhou, Yuchen Qian, Irina Calciu, Taesoo Kim, and Sanidhya Kashyap. 2022. Application-Informed Kernel Synchronization Primitives. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 667–682. https://www.usenix.org/conference/osdi22/presentation/park

[27] Pixie. 2023. Pixie. https://github.com/pixie-io/pixie

[28] Yocto Project. 2023. Yocto Project. https://www.yoctoproject.org/.

[29] Raspberry Pi Foundation. 2023. Raspberry Pi Software. https://www.raspberrypi.com/software.

[30] Dave Thaler and Poorna Gaddehosur. 2021. Making eBPF work on Windows. https://cloudblogs.microsoft.com/opensource/2021/05/10/making-ebpf-work-on-windows/

[31] Qiongwen Xu, Michael D. Wong, Tanvi Wagle, Srinivas Narayana, and Anirudh Sivaraman. 2021. Synthesizing Safe and Efficient Kernel Extensions for Packet Processing. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 50–64. https://doi.org/10.1145/3452296.3472929

[32] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. 2022. XRP: In-Kernel Storage Functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 375–393. https://www.usenix.org/conference/osdi22/presentation/zhong