A Case for Type-System Based Network Security

Hudson Ayers
hayers@stanford.edu
Stanford University

Armin Namavari
arminn@stanford.edu
Stanford University

Philip Levis
pal@cs.stanford.edu
Stanford University

ABSTRACT

We propose defining an operating system's network security in terms of rigorous, logical terms that are easy to express in a programming language: that language's type system. We describe how types and ownership in the Rust language can create capabilities that express useful security semantics, such as allowed remote addresses and ports. Simple capabilities can be checked at compile-time, while more complex ones are checked dynamically. The language's type safety ensures that these capabilities cannot be forged, duplicated, or corrupted. Beyond the benefits of easier auditing and reduced code size overhead, type-system based network security also improves security by requiring that developers explicitly distribute each instance of a capability, such that the path of least resistance follows the principle of least privilege.

CCS CONCEPTS

• Security and privacy → Network security; **• Computer systems organization → Embedded software**; **Embedded systems**.

KEYWORDS

Embedded Systems, Network Security, Type Safe Languages, Rust

1 INTRODUCTION

Cyber-physical systems are increasingly connecting to the Internet. This presents novel threats to their operation: no longer protected by walled gardens, they must defend against both targeted [4] as well as broad-scale [7] attacks. Insecure cyber-physical systems themselves pose a tremendous threat to the broader Internet; the Mirai botnet consisting of many different IoT devices was able to disable many large-scale Internet services for most of a day. [1]

We argue that network security should be a first-class concern in the design of next-generation cyber-physical operating systems. Rather than rely on a mix of configuration files, containers, firewalls, namespaces, and monitoring, security should be an integral part of every network operation within the kernel. To be able to communicate over a network using TCP on a certain port, a kernel service must provide proof that it is allowed to, in the form of a network capability. If the kernel service is compromised in some way, it can only communicate in the way its capability allows.

OS security policies today typically rely on configuration files or other out-of-band mechanisms, such as namespace configurations, `seccomp` settings, and file access permissions in Linux. We argue that network capabilities should instead be expressed in a rigorous, logical terms that are intuitive to express within the code itself: the language's type system. By writing OS services in a type-safe language and expressing capabilities within this language, OS developers and users can write and verify security invariants in the same language. Furthermore, writing these capabilities in the programming language removes the need for a file system or other out-of-band services a limited cyber-physical system may not have. Notably, while capability-based security is common in microkernel operating systems (such as seL4 [3]), in these systems networking functionality is typically implemented outside of the kernel. This design complicates enforcement of broad network-security policies from within the kernel, so we focus on monolithic-kernel systems.

We describe an initial prototype of this approach in the Tock operating system, which is designed to allow untrusted kernel extensions (called capsules) and untrusted applications written in C. [8] The Tock kernel is written in Rust, a type-safe programming language with performance and memory efficiency close to C. Rust's type safety ensures that only trusted code can create and configure network capabilities. However, untrusted code (e.g., kernel extensions) can safely use these capabilities to access network services, and untrusted code can check them as needed. In some cases, these checks can be at compile-time. For example, an in-kernel console can only make calls to the network if it has a reference to a network capability, and the compiler can in many cases verify whether such a capability exists at that point in code. In other cases, capabilities are verified at run time. For example, if the console operates on port 2323, the networking stack can check at compile time that its capability allows use of that port.

Expressing security policies in this way greatly reduces the amount of code that a developer must examine when adding new kernel code. A kernel extension requires a single line of code to instantiate it at boot. This line invokes 10-20 lines of trusted code that the developer must audit to check that it is requesting reasonable network capabilities. The extension itself can be thousands of lines of code, which the developer does not need to examine.

2 TYPE-SYSTEM BASED NETWORK SECURITY

Here, we describe our initial implementation of network security on Tock, and elaborate on how the Rust type-system has aided our goals. We want to emphasize that the point of this paper is not to preach our specific implementation as an ideal solution, but to broadly illustrate the value of type-system based network security.

To better secure embedded devices, we want the ability to implement verbose network security policies. These include policies such as those in Figure 1. We require a solution that gives us the flexibility to implement these diverse policies at low cost to flash/RAM requirements. Furthermore, verifying that these security policies are implemented for a given capsule should require looking at only a small portion of the codebase associated with these security policies, rather than requiring a full audit.

2.1 Rust, Tock, and Capabilities

NGOSCPS '19, April 15, 2019, Montreal, Canada

(1) “Capsule A cannot listen to the radio”
(2) “Capsule A cannot use the socket interface”
(3) “Capsule C can not send data without first encrypting it”
(4) “No two capsules can bind to the same UDP port”
(5) “Capsule B can only communicate with IP addresses 10.0.0.5 and 146.72.80.1”
(6) “Capsule D can talk to any node on the link-local network”

Figure 1: Example network security policies

```
macro_rules! create_capability {
    ($T:ty) => {{
        struct Cap;
        unsafe impl $T for Cap {}
        Cap
    };};
}
let process_management_capability =
     create_capability!(
        capabilities::ProcessManagementCapability);
```

Listing 1: Static Capability Example

Rust's type safety ensures at compile time that there are no null pointers, dangling pointers, buffer overruns, or data races. More strongly, it ensures that every every reference points to memory that stores the type denoted by the reference: pointers cannot be forged or corrupted. To support low-level code that must break Rust's restrictions (e.g., creating a reference to a CPU register from a hardcoded memory address), Rust provides the `unsafe` keyword. Code in an unsafe block can violate Rust's type safety, and functions marked unsafe can only be called from unsafe blocks.

The Tock operating system prioritizes safety by limiting unsafe code to the core kernel and low-level architecture-specific code. Most kernel code is in *capsules*, units of composition that can be included or excluded like external libraries. For example, the low-level SPI bus driver that must access memory-mapped IO registers can use the unsafe keyword (it is architecture specific), but a radio driver that uses the SPI bus is a capsule and cannot. Tock configures the Rust compiler to throw an error if a capsule uses unsafe code.

The Tock kernel uses an abstraction of “capabilities” to control access to particularly sensitive operations. Ex: a capsule need a `MemoryAllocationCapability` to allocate memory from a process, and a `ProcessManagementCapability` to start and stop processes. The code used to create these capabilities, along with an example instantiation, can be found in Listing 1. These capabilities have no storage and implement no functions but are marked unsafe. As a result, while they can be stored and passed by safe code (e.g., capsules), they can only be created by unsafe code. That is, capsules can use them, but trusted code within the core kernel must create them and pass them to the capsules. In practice, the kernel boot sequence creates these capabilities and passes them to capsules as needed. Because they have no functions or data, they are compiled away and add no overhead. At compile time, a capsule cannot be instantiated unless it is passed a valid capability, and the capsule can't call sensitive functions unless it has the required capability.

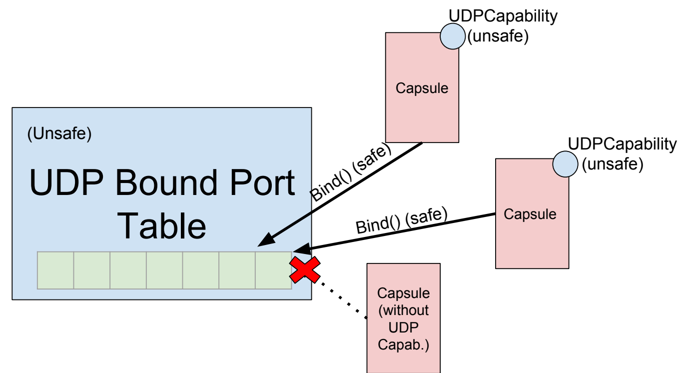

Figure 2: Port Table Binding via dynamic capabilities

2.2 Basic Design

Implementing network security on Tock requires more than the aforementioned static capabilities. While static capabilities are sufficient for basic network security policies, such as (1), (2), and (3) in Figure 1, static capabilities fail to address more complex policies such as (4), (5), and (6) in that list. Without creating thousands of individual capabilities (corresponding to each possible UDP port or possible external IP address), static capabilities can not handle such policies. To serve these more complex cases, we created 2 new types of dynamic capabilities.

The first type of dynamic capabilities rely on an `unsafe` data structure in the kernel which can only be modified by a set of safe functions. These functions can interact with and modify the data structure, enforcing policies such as (3) in Figure 1. Calling these functions requires capsules to pass a static capability. Because Rust's type checker will not allow code to compile if the arguments passed to a function are not the correct type, access to these in-kernel data structures is restricted at no runtime cost. For port binding, we use exactly this type of capability, and the underlying data structure is a small table containing `X` slots, where `X` is a pre-defined maximum number of networked capsules. Any time a capsule binds to a port, the port number is stored in the slot corresponding to the capsule with that ID. An example of this is contained in Figure 2.

The second type of dynamic capability we use is actually much more similar to static capabilities, except that the structs which implement the unsafe traits do have fields, which specify information related to the capability. The kernel initializes the fields of these capabilities with the appropriate parameters for the capsule it is being passed to. Because capabilities are passed as immutable references, the capsules cannot change the contents of the fields in the capability, and because the structs must implement an unsafe trait, capsules cannot create their own. Thus, these dynamic capabilities can only be constructed by the Kernel, but can contain a wide range of information, such as a list of acceptable external IP endpoints. The downside of these capabilities is that they have size, and cannot be evaluated purely at compile time. Thus, it is important to keep these capability structs relatively small. Nevertheless, between static capabilities and dynamic capabilities, we can express a wide range of securitiy policies at low cost.

2.3 Reducing the audit space

In most networked embedded systems, a network security audit requires reading the entire code base. While only portions of the application may use the networking stack, data structures associated with networking, such as those that describe port allocation or acceptable endpoints, can often be accessed (and modified) from anywhere in a monolithic application. Accordingly, verifying the network security of an external library loaded as a kernel extension would require auditing the entire library. While 'C' namespaces can sometimes be used to mitigate this, the monolithic structure of applications makes this difficult to apply in practice. Further, for systems without Rust's memory safety, any code might intentionally or unintentionally modify memory that contains security policies. Such a design makes the use of external libraries prohibitively time-expensive for security conscious developers.

The network capabilities we describe in the previous section substantially mitigate this concern. Because these type-based network capabilities can only be created or modified in the kernel, the only code which must be audited to ensure network security is the code in the kernel which creates and distributes these capabilities, along with the functions that check these capabilities. Even if some capsules perform capability checking themselves (such as a UDP capsule), the coverage required for a full system network security audit is substantially reduced. Even better, any kernel extension can be audited purely by examining the trusted code packaged in its initialization sequence. Typically this amounts to 10-20 lines of code, including creating capabilities. This is compared to the 500-2000 lines of code found in most capsules used in Tock. So long as the kernel does not distribute capabilities to this application, the content of the application does not matter — safe code cannot circumvent Rust's ownership rules and unforgeable pointers.

One apparent hole in this argument is that capabilities can still be moved between capsules, which might suggest that Capsule A could be surreptitiously passed Capability X by some Capsule A' that was legitametely given X. In an OS with a C linking model, this would be a significant issue. On Tock, however, this is only possible if Capsule A' has a reference to Capsule A, which is only possible if A' is passed a reference to A when A' is created. Accordingly, auditing the capabilities which A can obtain requires looking only at the kernel initialization code, not inside either capsule.

2.4 The benefits of explicit security

Traditionally, security policies such as those in Linux are second-class concerns, in that it is often easier to write code that ignores them and that often unspecified == unrestricted. We hypothesize that when developers can manipulate security objects in the same way they manipulate other objects in the type system, it is much easier to consider the impact of security decisions and much more natural to reason about the impact of different configurations. Further, we argue that this type-system based design makes it much more difficult to *compile* code that ignores security considerations, as function calls which require capabilities cannot compile without a handle to a capability of that type. Reducing the burden of implementing security is incredibly important, especially for low-margin embedded products unlikely to endure security reviews.

2.5 Open Problems

Realizing all of the benefits type-system based security will require addressing several open problems. First, a mechanism for extending kernel-service security requirements to applications needs to be implemented, such as a signed manifest on application data via public key encryption. Second, it has to be shown that dynamic capabilities similar to those we describe can cover an even broader range of security policies than those we have required so far. We can imagine information flow control or host-name based endpoint filtering as potential additional policies — exploring the limits of type-based capabilities will be a useful exercise.

3 RELATED WORK

Substantial related work exists in capability-based security, embedded system security and the use of type-safe languages to improve security. [9] provides an excellent comparison of capabilities and access control lists, an alternative to capability based security. [2] explores the use of the Rust type system to "enable...capabilities that cannot be implemented efficiently in traditional languages" to improve system security. [6] provides early insight into researchers grappling with insecure embedded systems, and argues that security needs to be elvated to the plane of metrics such as cost and performance during design. [10] details the dire need for secure embedded devices in cyber-physical systems, associated challenges, and argues in favor of cryptographically securing code using a co-processor. Finally, [5] illuminates how embedded network security policies can make simplifying assumptions, based around repetitive traffic patterns and limited communication partners.

REFERENCES

[1] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. 2017. Understanding the mirai botnet. In *({USENIX} Security 17)*. 1093–1110.

[2] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamari, and Leonid Ryzhyk. 2017. System Programming in Rust: Beyond Safety. *SIGOPS Oper. Syst. Rev.* 51, 1 (Sept. 2017), 94–99. https://doi.org/10.1145/3139645.3139660

[3] G. Klein et. al. 2009. seL4: formal verification of an OS kernel. In *SOSP*.

[4] Nicolas Falliere, Murchu, and Eric Chien. 2011. W32.Stuxnet Dossier. Symantec Security Response online report. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf

[5] J. Hong, A. Levy, L. Riliskis, and P. Levis. 2018. Don't Talk Unless I Say So! Securing the Internet of Things with Default-Off Networking. In *IoTDI 2018*. 117–128. https://doi.org/10.1109/IoTDI.2018.00021

[6] Paul Kocher, Ruby Lee, Gary McGraw, Anand Raghunathan, Srivaths Moderator-Ravi, and Srivaths Moderator-Ravi. 2004. Security as a new dimension in embedded system design. In *Proceedings of the 41st annual Design Automation Conference*. ACM, 753–760.

[7] Jakub Kroustek, Vladislav Iliushin, Anna Shirokova, Jan Neduchal, and Martin Hron. 2018. Torii botnet - not another Mirai variant. https://blog.avast.com/new-torii-botnet-threat-research

[8] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kb computer safely and efficiently. In *SOSP 2017*. ACM, 234–251.

[9] R. S. Sandhu and P. Samarati. 1994. Access control: principle and practice. *IEEE Communications Magazine* 32, 9 (Sep. 1994), 40–48. https://doi.org/10.1109/35.312842

[10] Michael Vai, David J. Whelihan, Benjamin R. Nahil, Daniil M. Utin, Sean R. O'Melia, and Roger I. Khazan. 2016. Secure Embedded Systems. *Lincoln Laboratory Journal* 22, 1 (2016), 110–122.