Privilege-separating Embedded Applications using WebAssembly in the Plat FIDO2 Security Key

by

Benjamin B. Kettle

S.B. in Electrical Engineering and Computer Science
Massachusetts Institute of Technology (2022)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2023

Authored by:	Benjamin B. Kettle Department of Electrical Engineering and Computer Science May 12, 2023
Certified by:	Anish Athalye Doctoral Candidate Thesis Supervisor
Certified by:	Nickolai Zeldovich Professor Thesis Supervisor
Certified by:	M. Frans Kaashoek Charles Piper Professor Thesis Supervisor
Accepted by:	Katrina LaCurts Chair, Master of Engineering Thesis Committee

Privilege-separating Embedded Applications using WebAssembly in the Plat FIDO2 Security Key

by

Benjamin B. Kettle

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 2023, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Plat is a FIDO2 security key that uses privilege separation to protect the application's private keys even if bugs are present in bug-prone parts of its codebase. Plat's design encapsulates drivers and parsers in sandboxes that are isolated from the secrets that are used to perform authentication.

To achieve privilege separation in the embedded context, Plat uses a new WebAssembly-based toolchain for ARM microcontrollers to implement and enforce isolation between individual components of an existing system without rewriting drivers and application code. This toolchain includes special support for device drivers, safely enabling isolated modules to access peripheral memory-mapped IO.

Plat's privilege-separation reduces the lines of code in the trusted code base by 60% from our 20,000-line reference implementation while adding only 319 new trusted lines. Plat's isolation strategy has acceptable performance overhead that does not prevent interactive use, with the slowest step of an authentication jumping from 277ms natively to 600ms when sandboxed.

Plat ensures the protection of its secret key, and thus the security of the accounts it authenticates, in the presence of several classes of bugs.

Thesis Supervisor: Anish Athalye
Title: Doctoral Candidate

Thesis Supervisor: Nickolai Zeldovich
Title: Professor

Thesis Supervisor: M. Frans Kaashoek
Title: Charles Piper Professor

Acknowledgments

The past year has flown by, but it would not have been nearly the great experience that it was without the amazing group of people that have gotten me to this point.

I am extremely grateful to Anish, who dedicated countless hours of his time over the past year and a half to teach, advise, mentor, and inspire me. He is incredibly generous, and I have learned much from him about everything from SMT solvers to White Mountains hiking.

It has been an honor to meet with Nickolai and Frans each week. Nickolai's incredible ability to follow a technical discussions anywhere it goes was instrumental in eventually settling on a research topic and he has played a critical role in shaping the direction of my research. Frans has been an excellent resource for all research-related questions. Nickolai and Frans have been superb advisors and I am lucky to have had them to guide me through my MEng.

I am glad to have been able to discuss teaching and research philosophy and priorities with Henry. His support over the last year has meant a lot.

Despite only being with the lab for a year, the wonderful students of PDOS have welcomed me into their home and have made for an excellent group of friends. I'll miss our incredibly varied—and sometimes rowdy—weekly discussions at group lunch and at "coffee hour."

As I transition from dedicating my life strictly to learning and begin to focus on learning while also doing other things, I am thankful to the many excellent teachers that have enabled this learning. From Mrs. Todd and Mrs. Maslow in elementary school and Mr. Campbell and Mr. Stineff in high school to the huge array of professors, instructors, and TAs that have fostered my interests at MIT, my teachers have made me who I am today.

Some of the most fulfilling work I have done at MIT has been as a teaching assistant. The professors and instructors with whom I have worked—Joe, Henry, Yael, Nickolai, and Srini—have all inspired in me a love for effective and caring teaching that I will carry with me into the future.

Finally, I am thankful to my parents and sister for instilling in me the love of learning and of nature that define me today, and to my girlfriend Naomi and the rest of my awesome friends for sharing the conversations, laughter, and experiences that have kept me sane over the past several years.

Contents

List of Figures

List of Tables

1

Introduction

1.1 Motivation

Passwords are used nearly everywhere on the web for authentication, from the smallest internet forum to the Internal Revenue Service. While convenient and easy to implement, passwords have a long history of security issues from weak passwords and common password reuse to phishing vulnerability and server database leaks [9, 28]. The FIDO2 standard, formed by a coalition of technology companies and quickly gaining momentum as a new authentication protocol [8], aims to solve many of these issues by replacing passwords with an automated public-key signature exchange. In doing so, FIDO2 avoids the need for users to manually remember and enter passwords, and thus eliminates many of passwords' inherent security risks, by reducing the security of web-based authentication to the security of a user's private key.

Deriving user identity from private key possession has the potential to provide strong security guarantees. But it also places a new requirement on users and their devices to manage and protect private keys. These keys are generally stored on dedicated *authenticator* devices like the popular YubiKey [56] rather than directly on the user's PC. In addition to a convenient portability, these security keys offer greatly improved security: they provide an API to the PC that allows it to obtain signatures over challenges to satisfy the authentication protocol, but do not allow the PC to read any part of the private key performing those signatures.

FIDO2 authenticators consist of much simpler hardware and software than a user's PC: instead of millions of lines of operating system code and a complex, highly speculative processor, security keys consist of tens of thousands of lines of code running on a simple embedded microcontroller. While this simplicity makes them less likely to have bugs, security keys—and similar devices like hardware cryptocurrency wallets, which perform the same basic function—are still susceptible to bugs that compromise security [48]. For example, the USB packet handler of the KeepKey hardware wallet had a missing length check that led to a buffer overflow attack, allowing an attacker to overwrite stack memory on the device [37]. A base64 parsing library used in the BitBox01 cryptocurrency wallet incorrectly handled invalid inputs and allowed an attacker to write to adjacent heap memory [35]. Even software by the YubiKey's manufacturer that implements U2F, a precursor to FIDO2, contained a bug that caused information to be leaked over USB [39]. Vulnerabilities like these compromise the security of the keys and credentials that consumers trust these devices to keep safe.

Though authenticators are simple compared to PCs, they still contain complex code, from the USB stack to message parsing code, that may contain bugs. While rewriting a project in a memory-safe language like Rust is likely to remove some bugs, the barrier to do so is high and, no matter the language, it is very difficult to remove all bugs from a system. Acknowledging this fact, this thesis builds Plat: a security key that aims to achieve some notion of security even in the presence of bugs and vulnerabilities.

1.2 Threat Model

Plat aims to provide security even when it is connected to a host PC that is compromised. Thus, the Plat threat model assumes that the connected host may be compromised and send arbitrary data over the USB interface it shares with Plat. This could include, among other behavior, disobeying the client-to-authenticator protocol (CTAP) that governs the interaction, sending packets out of order, or even sending malformed packets. We assume that bugs exploitable by this channel exist and that these vulnerabilities are severe enough—as many memory corruption bugs are—to allow arbitrary code execution.

Not included in our threat model are IO-based attacks that are not possible via software control over the host machine. For example, vulnerabilities that are exploited by sending out-of-range voltages over the USB D+ wire or by cutting the USB device's power precisely to enable a glitching attack are considered out of scope. While any embedded device is susceptible to hardware attacks such as RF emission analysis or even silicon-level techniques to reveal a secret stored in flash, these require the adversary to physically possess the embedded device. Our threat model excludes these vulnerabilities as well.

1.3 Security Goals

In the presence of remotely exploitable bugs included in our threat model, Plat aims to achieve several security goals:

G1. Secret Key Confidentiality: an attacker should not be able to learn any bits of the secret key.

G2. Secret Key Integrity: an attacker should not be able to modify or erase the secret key.

G3. User Signature Approval: every signature performed using the device secret key should be approved by the user physically interacting with the device (e.g. by pressing a button).

G4. Signature Counter Integrity: the signature counter described in the FIDO2 standard should monotonically increase with every signature performed by the authenticator and should not change unless a signature is performed.

1.4 Privilege Separation using WebAssembly

In order to limit the impact of bugs in the authenticator software and achieve our security goals in their presence, this thesis builds a FIDO2 security key that employs *privilege separation*. Privilege separation is a powerful technique that involves splitting up a codebase into several isolated pieces and giving each of those pieces only the permissions necessary to perform their intended actions. This way, a bug in any particular component allows the attacker control over only the parts of the system that the component could already access.

Desktop software can and does take advantage of process-based isolation and `seccomp`-enforced privilege separation to minimize the impact of bugs [24, 34]. This method of privilege separation relies on a virtual memory abstraction and a context switch for each new isolation domain and even for desktops, process-based isolation has high overhead. This overhead becomes significant for fine-grained isolation at the library level [30]. Further, authenticators like Plat are typically written without any underlying operating system and thus processes are altogether unavailable.

Software fault isolation (SFI) systems provide a solution by allowing isolation without hardware support, and software and cloud providers have turned to WebAssembly [20] as the de facto mechanism for software fault isolation. WebAssembly was created to allow performant execution of untrusted code on the web and solve problems that arose in past attempts like asm.js and Native Client [55]. To than end, WebAssembly's design goals included safe, fast, and portable language semantics as well as a safe and efficient representation. These goals turned out to be applicable far beyond the web: WebAssembly today serves as a portable and lightweight binary format that provides memory safety and control flow integrity without hardware support when used as a compilation target for any language. This has allowed not only the safe execution of untrusted code in the browser, but has also been used by cloud providers like Fastly who use WebAssembly to enable safe serverless edge compute with tenant code written in any language [16].

Table 1.1: CPU features of popular tokens & cryptocurrency wallets [2, 14, 21, 22, 40, 41, 42]. The underlying CPU architectures are parenthesized. Full MMU support in embedded devices is rare, and many popular FIDO2 tokens are implemented on platforms that do not support User/Kernel modes or memory isolation via an MPU.

Device	CPU	MMU?	MPU?	U/K Modes?
Feitan OpenSK	nRF52840 (C-M4)	✗	✓	✓
YubiKey Neo	LPC11U24 (C-M0) & A7005 (80C51)	✗	✗	✗
Google Titan	LPC11U24 (C-M0) & A7005 (80C51)	✗	✗	✗
Ledger Nano S	ST31H320 (C-M0)	✗	✗	✗
Trezor One	STM32F205RET6 (C-M3)	✗	✓	✓

The process-based isolation used on desktop systems relies on virtual memory provided by a Memory Management Unit (MMU). However, popular embedded processors such as the ARM Cortex M4 do not include an MMU, instead offering only a Memory Protection Unit (MPU) that allows setting permissions for a fixed number of aligned memory regions. Other processors like the Cortex M0 offer no memory protection at all. Even when an MPU is present, its restrictions mean that it is rarely used in practice [58]. Table 1.1 highlights that among common platforms for these hardware authenticators and similar devices, MPU support is uncommon. Even on devices that have them, MPUs alone do not enable privilege separation. Without the support of the process abstraction that is available on desktop machines, privilege separation on embedded devices requires toolchain support to compile and link each trust domain in a way that enables MPU use.

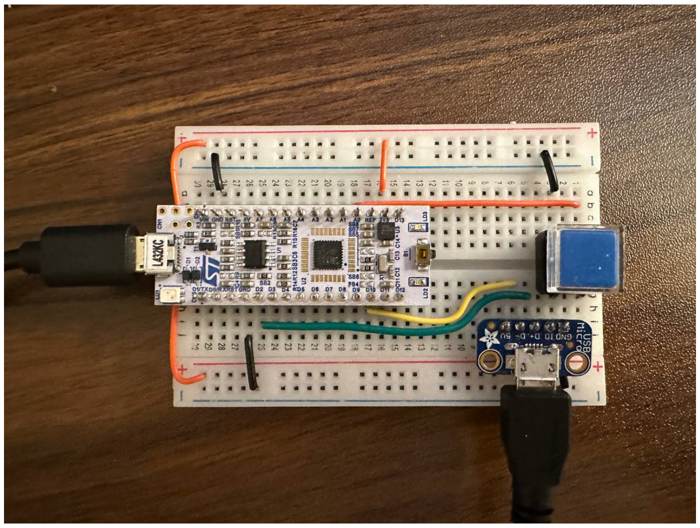
Figure 1-1: The breadboard used to prototype Plat. The hardware consists of an STM32L432KC development board, a push button, and a USB breakout board.

In order to implement privilege separation on a resource-constrained embedded device lacking sophisticated memory protection hardware and operating system abstractions, we turned to software fault isolation using WebAssembly.

1.5 Contributions

This thesis presents Plat, a FIDO2 security key that implements SFI-based privilege separation by applying WebAssembly to the embedded systems context. Plat's design contains bugs and limits the damage that they can cause.

Plat is based on the open-source Solokey security key software [43]. We modify their codebase in order to add privilege separation and use the same hardware as the commercial Solokey uses. Plat's prototyping hardware can be seen in Fig. 1-1.

This thesis makes several contributions:

- A toolchain for running WebAssembly modules on resource-constrained ARM microcontrollers.
- An approach and tool for providing WebAssembly modules with restricted access to peripheral MMIO registers, allowing the sandboxing of device drivers.
- Plat, a functioning privilege-separated FIDO2 token that provides goals G1 to G4 from Section 1.3 even in the presence of several classes of bugs.

This thesis discusses the implementation of Plat and presents solutions to challenges that arise when performing privilege separation of embedded devices like Plat.

2

Background & Related Work

2.1 Password Security

Used in everything from blanket forts created by children to websites that are the faces of billion-dollar companies, passwords have long been the de-facto method of access control. And indeed passwords can be quite powerful. In the case of passcodes to unlock a smartphone, for example, passwords work very well. The fact that the phone is a physical device and thus must be physically possessed by someone who is or is not the owner enables strict rate-limiting policies that restrict the allowable number of guesses enough that a 6-digit PIN is infeasible to guess while easy to memorize.

For authentication to online systems, however, passwords have long been known as a security weak point. As early as 1979, Morris and Thompson identified passwords as critical to the security of the original UNIX system and found that most user's passwords were "disappointing, except to the bad guy" [28], with 86% of passwords studied falling within a few trivially guessable classes. Decades later, little has changed: a 2007 study found that the average user has only 6 distinct passwords [17] and a study of passwords compromised in database leaks found that an attacker who tries only 10 guesses for each account on a service will be able to compromise 1% of all accounts [9]. Designed to be entered manually, passwords are trivially stolen: one survey found that 85% of surveyed organizations had been subject to at least one successful phishing attempt in 2022 and that 8% of the 7500 respondents had given their password to an untrustworthy source [33]. Passwords are not only inconvenient for users, but also for security professionals.

Many schemes have attempted to provide a replacement for passwords without these shortcomings [10]. Password managers like 1Password [1] provide a backwards-compatible improvement by providing software to automate password generation and entry, and many online services have implemented mandatory app- or SMS-based two-factor authentication in addition to a password. Despite industry interest, until recently no scheme has had success in replacing passwords altogether.

2.2 FIDO2 WebAuthn

WebAuthn, created by the FIDO Alliance [4], is a standard for authentication that aims solve these problems by using public-key cryptography to authenticate users in a computationally unforgeable way without sending any sensitive data over the network. With broad industry support from companies like Google and Apple [5], with Google supporting it as a primary login method for Google accounts [8], FIDO2 seems likely to largely replace passwords in the coming years.

FIDO2 WebAuthn provides strong security guarantees while avoiding vulnerability to phishing and database compromise by performing authentication via public-key signatures performed by software or hardware *authenticators*. Since these authenticators or not bound by human memory or cognition, they are able to authenticate using *private keys* that are random enough to be essentially unguessable and able to generate authentications specific to the exact URL of the server, or *relying party*, that the browser is connected to. Authenticators do not reuse keys to make it easier to remember then, and an authenticator cannot be fooled to authenticate a user to `google.com` when they are really visiting `gooogle.com`. Further, the cryptography behind public-key signatures enables the authenticator to prove to the relying party that it owns a certain private key without revealing the private key itself—no long-term sensitive data is sent over the network to the server. As shown in Fig. 2-1, the FIDO2 protocol dictates that the browser provides the current hostname to the authenticator, removing the risk of phishing attacks, and requires the authenticator to sign a random *challenge* that ensures each signature can be used to authenticate only once. Thanks to this computational authentication and public-key cryptography, FIDO2 WebAuthn obviates low-entropy passwords, password reuse, phishing, and sensitive database leaks.

FIDO2 provides strong security guarantees, but it also centralizes the security of a user's accounts onto a single authenticator, making the authenticator a sensitive target for attacks. The security of this authenticator is crucially important to the security of FIDO2 overall.

While both *platform authenticators*—built in to a general-purpose device like

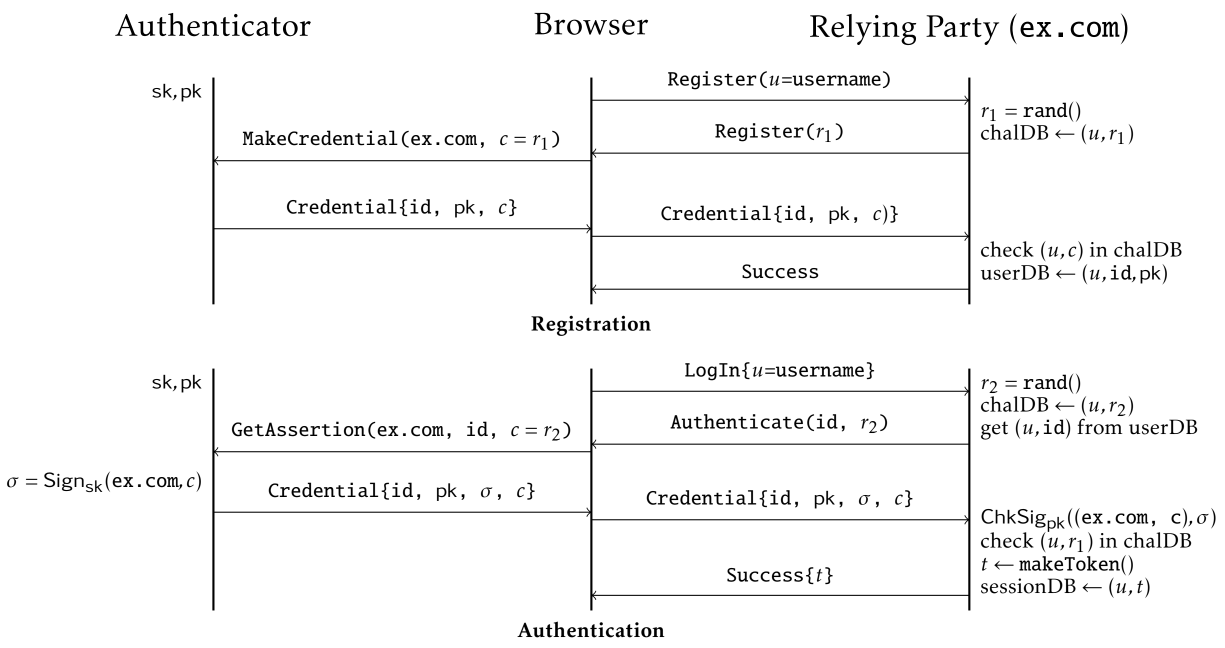

Figure 2-1: The FIDO2 WebAuthn registration and authentication protocols (simplified). Importantly, the Relying Party verifies that the authenticator's signature covers its own hostname and challenge (r_1), preventing phishing and man-in-the-middle attacks. The signature serves as a single-use token issued by the authenticator, which the server validates before issuing the browser a session token. No long-term secrets are exchanged over the network.

a laptop or smartphone—and *roaming authenticators*—implemented as a separate portable device—are supported by the FIDO2 spec, authenticators today are generally implemented as security keys. The YubiKey in Fig. 2-2, for example, is a FIDO2 security key that has sold over 22 million units [56]. In addition to a familiar lock-and-key mental model and convenient portability across devices, implementing an authenticator on separate hardware provides great security benefits by turning the authenticator into a *hardware security module*.

2.3 Hardware Security Modules

Security vulnerabilities are rooted in bugs. While bugs are likely in all software, it is much easier to reason about and test a small codebase than a large one for correctness. For this reason, in production systems many security-critical operations are factored out onto dedicated devices called hardware security modules (HSMs). For instance, the Let's Encrypt certificate authority stores its private keys on HSMs that perform signatures but do not reveal the keys to an attacker [3] and Apple uses

Figure 2-2: The YubiKey, the most popular commercial FIDO2 security key, comes in a small USB-stick form factor and supports USB and NFC protocols for use with PCs and mobile devices.

HSMs to enforce rate limiting for recovery of iCloud backups [25]. Isolating this sensitive code onto separate physical hardware separates it from the millions of lines of operating system and software code—and the bugs that are likely present in them—running on a typical server. By carefully auditing the software running on these HSMs, implementers can hope to remove most bugs.

HSMs greatly reduce their bug exposure by removing the operating system and interfaces like networking. But HSMs still need to provide a small subset of the functionality that an operating system would provide, and writing this low-level code is difficult. Despite careful review, HSMs have suffered from bugs [15, 48, 57].

2.4 Embedded Security

Bugs in embedded devices are rarer than bugs in desktop and server systems due to sheer size—embedded systems have far fewer lines of code than desktop operating systems and software—but embedded systems still contain lots of code that might be untrustworthy. Not only is most embedded software written in C, a language that lacks safety features to prevent memory corruption and logic bugs, but embedded software is often complex: communicating with peripherals such as USB often involve packet parsing, bit wrangling to interface with control registers, and other bug-prone tasks. And, in contrast to the desktop environment, embedded systems make use of a wide range of processors, each supporting a different set of peripherals and even multiple architectures. This means that embedded libraries are target-specific and lack the widespread testing that desktop libraries enjoy. Past work has sought to provide solutions to these difficulties.

TinyOS [26] provides an environment and programming model to simplify programming severely resource-constrained devices, including device drivers and an efficient operating system implementation to allow for task-based programming—without much focus on security. For the chips it supports, Tock [27] provides a full operating system for embedded devices with support for processes ("applications") isolated by the MPU and, importantly, for memory-safe kernel modules and drivers ("capsules") with safety enforced by the Rust language. Tock prevents memory errors in kernel space (via Rust), but applications can be written in any language and themselves are not privilege separated—Tock prevents a buffer overflow in the kernel space UART driver, but does not prevent logic errors on the application level nor a buffer overflow in an application-level packet parser. To use a peripheral with Tock, its driver must be rewritten in Rust and in the Tock programming model and application code must be written to use Tock system calls—adapting an existing project to use Tock would be a significant undertaking.

OpenSK [18] implements a FIDO2 token written as a single Rust Tock OS application, rewriting common CTAP protocol drivers and other code in Rust to achieve memory safety within the application in addition to the kernel space safety that Tock provides.

Formal verification provides a more satisfying solution—full adherence to a higher-level specification—for simple applications. Notary [7] implements a device capable of running several security-key-like approval agents while ensuring that each agent is isolated from the others by formally verifying a full CPU reset procedure that runs between each isolation domain, but does not contain the damage of possible bugs within a single agent. Knox [6] provides a security definition for hardware security modules in general and a framework for formally verifying entire implementations of HSMs that encompass both hardware and software. Provided that the verification framework and specification are correct, this allows ensuring with certainty that simple HSMs are completely bug-free. However, in its current state Knox's verification approach scales only to simple HSMs—Knox's examples use UART for I/O rather than USB—and supports only simple cryptography like symmetric-key encryption and hashing: public-key operations like those required for a FIDO2 token are not supported.

While valuable tools, none of these approaches provide a mechanism for containing damage from bugs within existing code without requiring substantial rewrites of drivers and application code.

2.4.1 Driver Security

In and out of the embedded context, device drivers are a common source of buggy code and security vulnerabilities, particularly since even in sophisticated operating systems like Linux they are run in the trusted kernel. Research has long tried to move drivers out of the trusted code base. The Nexus operating system, which executes drivers in user space, uses a Reference Validation Mechanism [54] to define valid user space driver behavior using an expressive specification language and ensure at runtime that a driver is behaving according to that spec, monitoring for and preventing bugs that might allow a driver to escape its user-level permissions or affect overall system availability. The Nexus RVM is able to monitor and validate drivers for complex devices like an e1000 Ethernet card. KSplit [23] provides a framework for isolating unmodified device drivers from the rest of the kernel using software fault isolation, enabling automatic analysis of driver code and generation of glue code to copy state between the kernel and driver process as necessary. KSplit's framework protects the kernel state from memory corruption and other bugs in device drivers while avoiding unnecessary state copying to minimize overhead. Both tools are very powerful, but are built for the desktop environment and introduce large dependencies.

2.5 Privilege Separation

Privilege separation is a powerful technique for achieving security guarantees even in the presence of bugs. A typical system without privilege separation runs application code as a single unit with all parts equally privileged while including a huge range of libraries that may be poorly tested or even malicious. Though a base 64 parsing library may not require access to the file system or the network, for example, a typical application would allow it to access the file system if it included code to do so or a bug in its code allowed an attacker to instruct it to. Privilege separation splits an application into several *sandboxed* components whose access to the rest of the system is limited. Sandboxed components have their own memory space and can call only functions within the component. Sandboxed components can interact with the rest of the system only through explicitly defined APIs. The APIs provided to each component should follow the *principle of least privilege*, giving each sandboxed component only the permissions required to perform the task that the application needs of it.

In a well-privilege-separated system, the base 64 parser above might be in its own sandbox with a very simple API, perhaps a single `decode(...)` function. The library requires no access to system functions at all, and thus privilege separation is able to provide strong guarantees about the system's exposure to bugs in the parser: even if an attacker is able to supply an argument to the parser that causes arbitrary code execution—a severe but not unlikely vulnerability—the worst that the attacker can do is cause the parser library to supply incorrect results. Since the parser is inside a sandbox, its only interface to the outside world is through the API defined for the sandbox. The sandbox will ensure that the module cannot corrupt state or call functions that are not explicitly provided to it, preventing damage. Since removing all bugs from an application is difficult, privilege separation provides an effective layer of defense.

One of the main challenges in privilege-separating a system effectively is isolating the components from each other at a fine enough granularity to improve security but without negatively impacting performance. Privilege separation has been implemented in several production systems with different isolation strategies. OKWS [24] implements effective privilege separation of a web server where each process has access only to strictly necessary resources, and OpenSSH [34] does so in the widely used SSH server. Both OKWS and OpenSSH use Linux processes to isolate components from each other. Seeking better performance and more fine-grained isolation, RLBox [30] uses WebAssembly to achieve isolation and privilege separation at the individual library level in the Firefox renderer, compiling untrusted libraries from C to WebAssembly and back into C before linking them with the rest of the Firefox code to take advantage of WebAssembly's memory safety and control flow guarantees. This allows Firefox to avoid the overhead of context switching while still protecting the main renderer process from bugs in untrusted libraries.

2.6 WebAssembly

WebAssembly was designed as a performant and efficient compilation target for use on the web [19]. Designed for running code from untrusted web servers, strong isolation is critical for WebAssembly—as it is for JavaScript—and WebAssembly is designed as a sandboxed language: by WebAssembly semantics, WebAssembly programs run independently from the *host environment* that is running them, and it is not possible to write a WebAssembly program that influences the behavior of the host environment except as explicitly allowed by APIs provided by the host environment. While WebAssembly instructions provide byte-level load and store access, enabling fast and simple translation to native code, WebAssembly programs have access to their own memory space that is isolated from the rest of the host system's memory and the WebAssembly language allows sandboxed code to jump only to other code within the sandbox. These memory safety and control flow integrity semantics ensure that bugs or malicious code within a module cannot compromise the host environment.

WebAssembly is well-supported by browser vendors like Google, Mozilla, and Microsoft, and has been deployed by many web services that do CPU-heavy processing in the browser. Figma, an online design software provider, saw a 3x improvement in document load times after switching from JavaScript to WebAssembly [50]. But WebAssembly's performance and strong security guarantees, along with its design as a universal compilation target for many source languages, has made it attractive beyond the browser as well. CDNs like Fastly and Cloudflare, who run customer code on servers distributed across the globe, have found WebAssembly convenient for providing isolation between customers without the overhead of VMs or containers [16, 49].

WebAssembly semantics are described in terms of a virtual stack machine with a dedicated linear memory region. To execute WebAssembly code on today's register-based processors, WebAssembly code must be translated into machine instructions while preserving WebAssembly's memory safety and control flow integrity semantics. This translation can be done either ahead of time via a compiler, like V8's JIT compiler or Wasmtime, or at runtime by an interpreter like wasm3 [51]; in either case, the integrity of the sandbox depends on the correctness of the translator—a bug in the compiler or runtime could be exploited by module code to violate WebAssembly's guarantees. vWasm and rWasm [11] seek to guarantee the correctness of a WebAssembly compiler by relying on formal methods and, with better performance, the correctness and safety of safe Rust code. HFI [29] explores an extension to x86 that extends hardware support for WebAssembly and other SFI schemes.

2.6.1 WebAssembly on Embedded Systems

WebAssembly's safety guarantees and good support as a compilation target make it attractive for use in embedded systems as well, especially as WebAssembly does not require the hardware support of other isolation mechanisms like process-based isolation. eWasm [32] considers many of the challenges that WebAssembly presents for embedded devices given its design for 64-bit desktop machines and presents a performance-focused ahead-of-time compiler and runtime for Wasm modules on embedded systems and an analysis of possible changes that could make WebAssembly easier to use on embedded systems. Wasm3 [51] is an interpreter designed for embedded platforms and has dealt with many of the same limitations that eWasm and Plat have.

3

Design Overview

Plat adds privilege separation to Solokey [43], an open-source FIDO2 security key that is commercially available, while reusing nearly all of its code. Solokey is implemented as a monolithic C application; Plat divides this monolith into several trust domains called *modules*, each with only the privileges required to do their job. Choosing good module boundaries is critical for privilege separation to be effective.

3.1 Effective Module Boundaries

Modules provide an API exposing certain functions to the *host environment* in which the module runs, have access to a limited set of MMIO peripherals depending on the module's function, and can call a set of external function that the host environment explicitly exposes to the module. Besides this limited interaction with the host, modules are designed to sandbox the code inside: they prevent side effects outside of the module regardless of the code that runs inside the module.

By limiting the functions exposed to a module and the MMIO peripherals the module has access to, we can limit a module's capability to cause harm. In the original Solokey code, for example, chip-specific code running as part of the USB driver has exactly the same permission level as highly tested cryptography code that accesses sensitive secret keys: a buffer overflow within the USB stack could be exploited to overwrite the secret key. By sandboxing code in modules and restricting their access to the host environment, we can ensure that the USB driver can access only the USB peripheral and no other memory. Even if there is a bug in the USB driver code that leads to arbitrary code execution, an attacker's capabilities will be limited to those explicitly provided by the host: assuming the sandboxing system is correct, the worst an attacker will be able to do is send messages of their choice over USB.

Since permissions for access to the host environment are defined on a per-module basis, the definition of module boundaries crucially defines the effectiveness of privilege separation. First, module boundaries should be chosen such that bug-prone operations are isolated from those that are essential to achieve a system's security goals. Placing the entire application in a single module, for example, will provide negligible security benefit since no isolation is provided to limit the impact of a bug. Second, module boundaries should be chosen to require only a narrow API across the module boundary. Every function that a module imports is a hole in the otherwise secure sandbox and introduces opportunities for bugs, so the integrity of the sandbox relies on the correctness of safety checks at every function exposed to the module. Further, since modules each have their own memory space and function arguments must be copied in and out of each module, complex data structures are difficult to pass across module boundaries. It is essential to limit the number and complexity of functions that cross this boundary.

Privilege separation boundaries in Plat are designed to limit the damage that can be cause by bugs that we anticipate.

3.2 Designing Trust Domains in Plat

While all code may potentially have bugs, complex code and less heavily used code is much more likely to contain bugs than relatively simple code or code that is in widespread use. Driver code, which not only necessarily performs complicated bit manipulation and parsing but is also written individually for every peripheral device, is both complex and in limited use. Indeed, driver code is responsible for a majority of the bugs in the Linux kernel [12]. In order to achieve our security goals of G1 Secret Key Confidentiality, G2 Secret Key Integrity, G3 User Signature Approval, and G4 Counter Integrity, we designed Plat with this expected bug distribution in mind.

In particular, we expected bugs to be most likely in the USB stack of Plat, in the CBOR parser used to encode and decode messages sent over USB, and in the code responsible for parsing those decoded messages. Libraries like USB stacks and parsers have been responsible for bugs in the past, these particular ones are not very widely used, and these are simply the most complicated parts of the code base.

To achieve our security goals, this bug-prone code must be isolated with sandboxes from the state that we seek to protect: the master secret from which private

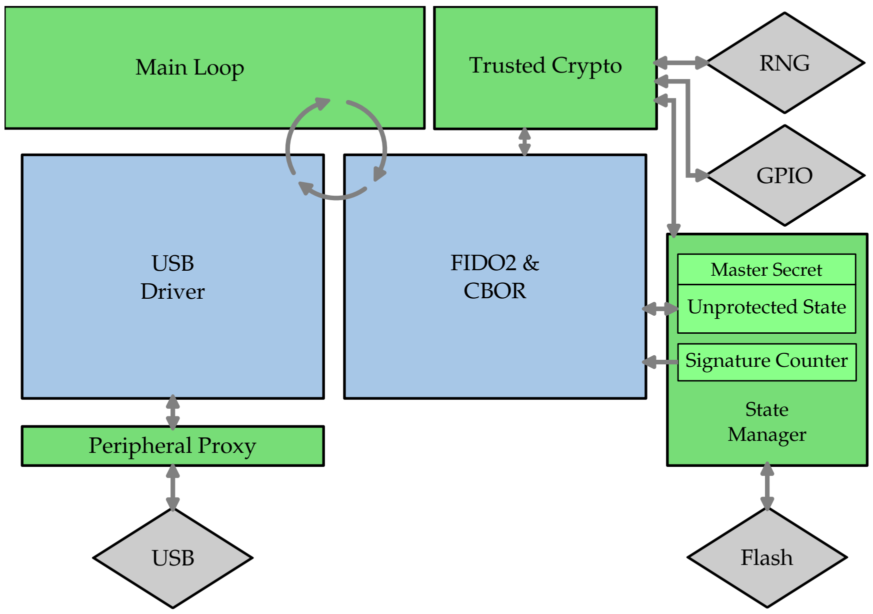

Figure 3-1: We sandbox bug-prone driver code in the USB stack and complex FIDO2 parsing code, isolating each as much as possible from the cryptography code that must access the secret key: neither module can access the master secret, and only the FIDO2 module has any access to trusted crypto functionality and to the managed state. In general, packets flow from the USB module to the FIDO2 module, which generates a response and sends it to the USB module again. Sandboxed modules are in blue, components that execute natively are in green, and hardware peripherals are in gray.

keys are derived. This way, bugs in the bug-prone code can cause arbitrary damage within a sandbox but an attacker that exploits one of these bugs will be unable to escape the sandbox and read or modify the master secret.

As seen in Fig. 3-1, we protect this secret by creating two sandboxed modules: one that contains the USB driver code and another that contains the FIDO2 logic and CBOR parser that decodes, parses, and encodes messages from and to the host PC. This ensures that even if bugs in those modules allow memory corruption and arbitrary code execution, an attacker cannot modify memory or run code outside of each module. We limit damage even within the bug-prone codebase by using two modules instead of one: if the USB driver has a bug, an attacker will still be unable to get even a signature from the Trusted Cryptography functions, and if the FIDO2 module has a bug the attacker will still be subject to FIDO2's limited API to the USB driver and the hardware. Plat implements a toolchain, discussed in Chapter 4, to isolate these modules using WebAssembly.

Importantly, the cryptography code that accesses the master secret is not included in either of these modules. Instead, any code that requires access to the secret key—what we call "trusted crypto"—remains directly compiled to native code without any sandboxing as discussed in Section 6.1.

Very high-level control flow—the main loop of the device—is also compiled directly to native code. Seen in Fig. 3-2, this main loop continually forwards packets that are read from the USB HID module to the FIDO2 module for processing. The FIDO2 module sends responses to these packets by calling the USB module's `send` function directly.

```
while(1)
{
    ... // send heartbeats if necessary

    if (usbhid_recv(hidmsg) > 0) // reads from USB HID module
    {
        ctaphid_handle_packet(hidmsg); // sends packet to FIDO2 module
        memset(hidmsg, 0, sizeof(hidmsg));
    }
    ctaphid_check_timeouts();
}
```

Figure 3-2: The main loop of the SoloKey code simply reads HID packets from USB and passes them to the FIDO2 module for processing. In response, the FIDO2 module makes calls to `usbhid_send` to return packets to the host.

The module containing the USB driver requires access to the USB peripheral's memory-mapped IO. To enable this, Plat implements a *peripheral proxy*, described in Chapter 5, that validates and permits access to peripheral memory by each module.

The master secret and other non-confidential state such as the authenticator's PIN code and the signature counter are stored persistently in Flash memory. Because of this, allowing the FIDO2 module to directly access to the Flash peripheral MMIO via the peripheral proxy would allow an attacker to violate our security goals: they could simply read the secret from Flash. Instead, Plat also implements a *state manager*, detailed in Section 6.2, that allows indirect access to the persistent state while preventing the FIDO2 module from accessing the master secret.

This design also allows Plat to ensure G3 User Signature Approval and G4 Counter Integrity. The trusted cryptography code includes checks for user presence and code to increment the signature counter, ensuring that bugs within the module cannot interfere with these processes. Plat provides read-only access to the signature counter to the FIDO2 module, ensuring that an attacker cannot change the underlying flash storage.

More fine-grained isolation, or other divisions altogether, are possible, but every isolation level requires a self-contained module with a clearly defined API. The module choice that Plat uses allows a narrow API between modules by following existing divisions in the Solokey source where possible and allows us to sandbox likely sources of bugs while protecting critical cryptography code and state.

4
Achieving Isolation with WebAssembly

We use WebAssembly to sandbox the C code in each module, taking advantage of its active development, its widespread support as a compilation target, and, most importantly, its strong isolation guarantees. While WebAssembly is a great fit for the conceptual needs of our application and its design as a multi-platform instruction format [19] means that it can run on many targets, it was not designed to run on resource-constrained embedded microcontrollers. This thesis presents a toolchain to sandbox C code by compiling it to WebAssembly and running the WebAssembly module on an embedded ARM processor, achieving the isolation that we require to implement privilege separation. We achieve this by compiling the WebAssembly to native code using a trusted compiler and compiling the generated C source to native code to generate a single binary.

4.1 Running WebAssembly on ARM

Despite the reality that WebAssembly was not designed with small embedded processors as a target, there is sufficient platform-agnostic tooling, in addition to limited prior work [32, 51] to develop a toolchain. This toolchain combines source code from the different modules and from trusted code (that will run natively with full permissions) into a single binary that can be flashed onto the microcontroller. Assuming that the tools and compilers used to generate it are correct, the resulting binary will enforce WebAssembly's isolation guarantees for the sandboxed modules.

In order to allow the reuse of existing SoloKey code and to simplify Plat's toolchain, Plat uses C as the programming language for both module and trusted

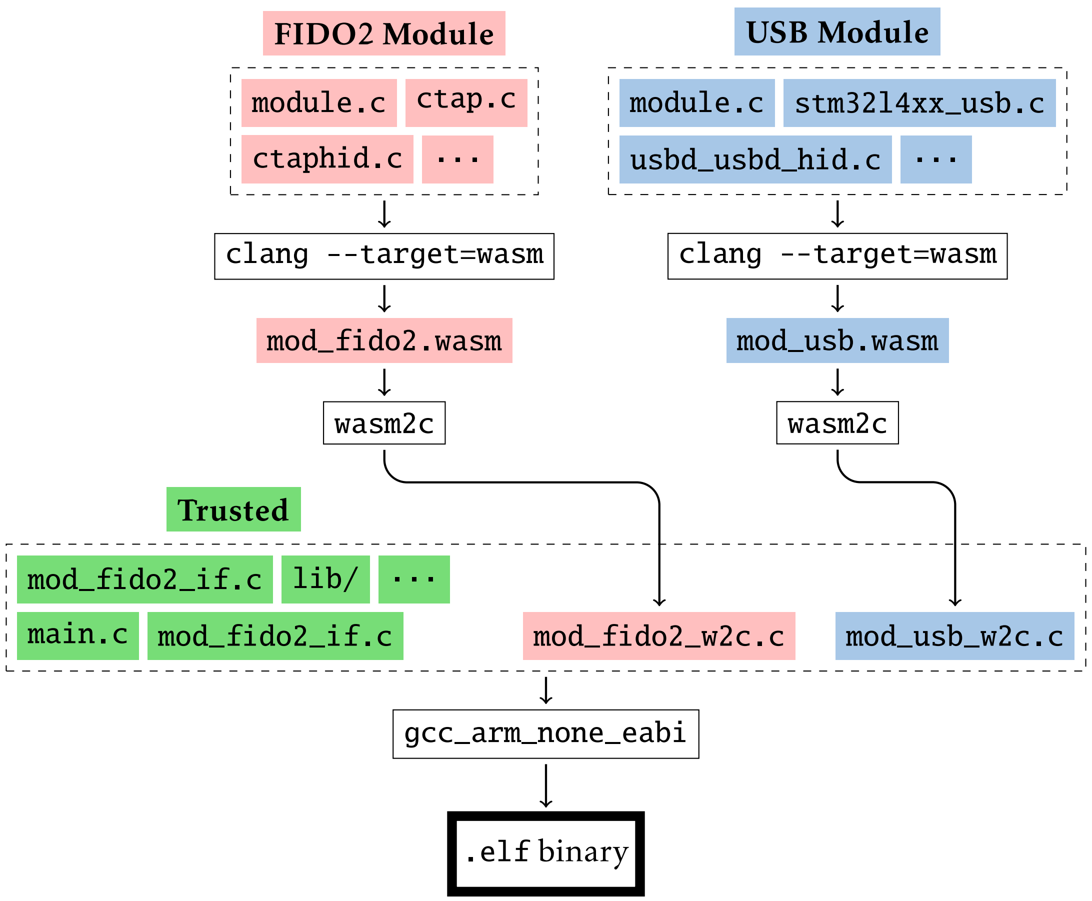

Figure 4-1: The toolchain to combine module and trusted source into a single ARM binary. C source code for each module is first compiled to WebAssembly using `clang`, then back into C with safeguards in place using `wasm2c`. Trusted source code is finally compiled together with sandboxed source code (files ending with `_w2c.c`) using `gcc` to produce a single binary.

source code, including the sandboxing infrastructure itself. As shown in Fig. 4-1, Plat's toolchain uses C at multiple steps, as both a source language and a compilation target.

To isolate the modules, the first step of the compilation process is to compile the C module source code into a WebAssembly module. To do this, our system uses the WebAssembly backend for LLVM together with `clang`. Most initial development work on WebAssembly proofs-of-concept have used Rust—whose compiler is LLVM-based—as a source language, so this backend is quite well supported. Compiling C code in this way generates platform-agnostic WebAssembly binaries: the `.wasm` files in Fig. 4-1.

The run the generated WebAssembly binaries, we use `wasm2c` to generate C code from each WebAssembly binary. Provided that we trust the `wasm2c` compiler, this generated C implements WebAssembly semantics and provides WebAssembly's safety guarantees, so can be compiled along with the rest of the trusted C codebase. This allows us to avoid the code size overhead of a universal WebAssembly interpreter like Wasm3 [51] alongside already large WebAssembly binaries and allows simpler compilation than would be possible using a dedicated WebAssembly to ARM compiler like aWsm [32]. `wasm2c` generates C code, shown as the `*_w2c.c` files in Fig. 4-1, that can be compiled along with the non-sandboxed trusted portions of the C codebase.

After using `clang` and `wasm2c` to generate C files encoding the sandboxed modules, all that is left is a C codebase comprising the generated C and the trusted C source that will not be sandboxed. The final step is to compile this C code into a binary that can be flashed onto the microcontroller. For this, we use the standard `gcc-arm-none-eabi` compiler.

4.2 Linking WebAssembly Modules

Plat's FIDO2 module is structured such that its entry point is a single function, `ctaphid_handle_packet`. This entry point performs all necessary processing on the received packets and, as needed, calls `usbhid_send` to transmit packets back to the host PC in response. In Plat's privilege-separated design, the main loop that calls this entry point is part of the host environment, so the FIDO2 module needs to export `ctaphid_handle_packet` so that the host environment can call it. The `usbhid_send` function is implemented and exported by the USB HID module, and thus the FIDO2 module must import that function from the USB HID module. WebAssembly supports imports and exports to the host environment by default, but does not support linking between modules. Native support for this inter-module linking with support for complex type conversion at module boundaries is planned as part of the WebAssembly component model proposal [52], but is very much a work in progress and is still in the proposal stage with no full implementations. Plat requires only basic linking between modules, which is simple to build on top of `wasm2c`.

In order to use imported functions within and export functions from a module, we take advantage of LLVM's support for `import_name` and `export_name` attributes on C functions. By marking declarations and implementations as in Fig. 4-2, we instruct LLVM to define the relevant functions as imports and exports in the generated WebAssembly. Importantly, imports and exports must be declared statically and imports must be explicitly wired up by the host: a module cannot import a new function at runtime, and functions that a module declares as an import but the host does not define will lead to an compilation error.

```
__attribute__((import_name("usbhid_send")))
void usbhid_send(uint8_t * msg);

__attribute__((export_name("ctaphid_handle_packet")))
void ctaphid_handle_packet(uint8_t * buf);

static void ctaphid_write(CTAPHID_WRITE_BUFFER * wb, void * _data, int
    len)
{
      ...
            usbhid_send(wb->buf);
      ...
}
```

`ctaphid.c` (module source)

```
static void w2c_ctaphid_write(Z_fido2_instance_t* instance, u32 w2c_p0,
    u32 w2c_p1, u32 w2c_p2) {
  ...
  w2c_i2 -= w2c_i3;
  w2c_i0 = w2c_memset(instance, w2c_i0, w2c_i1, w2c_i2);
  w2c_i0 = w2c_p0;
  (*Z_envZ_usbhid_send)(instance->Z_env_instance, w2c_i0);
  ...
}

/* export: 'ctaphid_handle_packet' */
u32 Z_fido2Z_ctaphid_handle_packet(Z_fido2_instance_t* instance, u32
    w2c_p0) {
  // this function calls w2c_ctaphid_write to send packets to the host
  return w2c_ctaphid_handle_packet(instance, w2c_p0);
}
```

`fido2_w2c.c` (generated by `wasm2c`)

Figure 4-2: Wasm2c's handling of module imports and exports for the FIDO2 module. The FIDO2 module imports **`usbhid_send`** to send USB packets to the host PC. When compiled with `clang` and `was2c`, this code generates calls to a function `Z_envZ_usbhid_send` that must be implemented by the host environment.

The `wasm2c` compiler handles module imports and exports by defining mangled versions of the import and export names. To call a function exported by a module, the host environment calls the generated name, and to allow a module to import a function, the host must define a function with the name generated by `wasm2c`. Fig. 4-2 shows a snippet of `ctaphid.c`, one of the source files for the

FIDO2 module, that declares `usbhid_send` to be an import from the environment and `ctaphid_handle_packet` to be exported to the environment. It also defines `ctaphid_write`, which uses the imported `usbhid_send` to send data to the host PC. When the FIDO2 module is compiled to a WebAssembly binary using `clang` and then to sandboxed C code with `wasm2c`, `wasm2c` produces C code, as shown in `fido2_w2c.c` in the figure, that makes a call to `Z_envZ_usbhid_send`. In order for the host to run this module, it must define a function with this name to allow the module to import `usbhid_send`. With this import defined, the host environment can make calls to the generated function `Z_fido2Z_ctaphid_handle_packet` to call the entry point of the module.

Fig. 4-2 demonstrates that `wasm2c` supports single-module imports and exports cleanly. For our privilege-separation use case, however, we require that modules can call *each other*. WebAssembly and `wasm2c` do not support directly linking modules. However, it is easy to work around this with a bit of glue code.

The FIDO2 module in Fig. 4-2 imports `usbhid_send` from the environment. Since this function is implemented in the USB HID module in Plat, this import needs to be provided by the USB HID module. As shown in Fig. 4-3, we can enable this by defining the `Z_envZ_usbhid_send` function called by the generated code to call the `Z_usb_hidZ_usbhid_send` function that is exported by the USB HID module.

```
#define PKT_BUF_SIZE 64
void Z_envZ_usbhid_send(struct Z_env_instance_t* env, u32 msgGuest) {
    return usbhid_send(translateGuestOffset(env, msgGuest, PKT_BUF_SIZE)
   );
}

void usbhid_send(uint8_t * msg) {
  memcpy(inBuffer, msg, INOUT_BUF_SIZE);
  Z_usb_hidZ_usbhid_send(&usb_instance,,translateHostAddr(&usb_env, inBuffer));
}
```

Figure 4-3: To allow the FIDO2 module to import `usbhid_send` from the USB HID module, we define FIDO2's import to call USB HID's export. Note that `usb_instance` is a pointer to the `Z_usb_hid_instance_t` struct for the USB HID module. We store pointers to these structs globally for every module.

This adds a bit of extra implementation work since we need to define a glue layer that maps the import of one function to an export of another, but is otherwise quite simple. By defining functions in this way, we can enable inter-module linking in `wasm2c` despite a lack of explicit support.

4.3 Safely Handling Pointer Arguments

WebAssembly natively supports only integer arguments to imported and exported functions. For any nontrivial application, this is not sufficient: it is necessary to pass at least serialized blobs of data in and out of a WebAssembly module. In our approach, we do so by passing pointers into module functions directly and having modules pass pointers back. Modules, however, cannot access all of physical memory, and when the C code inside the module gets compiled, memory from its perspective begins at the beginning of module memory: a given address refers to entirely different physical memory locations depending on whether it is used inside or outside of the sandbox. The module requires pointers into its own memory space, while the host can use only pointers relative to all of physical memory. We refer to pointers into module memory as *offsets* and pointers into physical memory as *addresses*. These reference frame differences cause two distinct challenges when calling exported module functions and when providing functions to modules for import.

To handle both of these challenges, Plat implements two translation functions, `translateHostAddr` and `translateGuestOffset`, shown in Fig. 4-4, to convert physical memory addresses into module offsets and module offsets into physical memory addresses respectively.

The former is used for the host to call exported module functions, since data that the host would like to pass into a guest module must reside inside the guest's memory space. As shown in Fig. 4-5, we define a simple heap allocator inside each module that allows the host environment to dynamically allocate a memory region inside module memory and receive the offset within module memory at which the allocated region begins. By converting valid module offsets into physical memory addresses, `translateHostAddr` allows the host to convert the offset of the allocated memory region inside the module into a valid physical memory address and copy data into the module. It can then pass the offset as an argument into the module function, where it will be interpreted as a pointer. Plat wraps each exported function in an *interface function*, as shown in Fig. 4-5, to abstract away this translation complexity for calling code in the host environment. With this interface function in place, the host code calls the exported module function exactly the same way that it would have if the function were not in a module at all.

The latter translation function is necessary when a module calls an imported function that takes pointer arguments. The calling module will supply offsets into

```c
typedef struct Z_env_instance_t {
  int module_class;
  uint8_t** memory_base;
  uint32_t* memory_length;
} Z_env_instance_t;
```

```c
uint32_t translateHostAddr(Z_env_instance_t* env, void *hostAddr) {
  if (
    hostAddr < *(env->memory_base) ||
    hostAddr >= (*(env->memory_base) + *(env->memory_length))
  ) {
    // error, host addr not in sandbox
    while (1);
  } else {
    return (uint32_t) hostAddr - *(env->memory_base);
  }
}
```

```c
void * translateGuestOffset(Z_env_instance_t* env, uint32_t guestOffset,
    uint32_t len) {
  if (
      guestOffset < *(env->memory_length) && // beginning in sandbox
      guestOffset + len >= guestOffset && // len doesn't cause overflow
      guestOffset + len < *(env->memory_length) // end in sandbox
  ) {
    // safe to translate
    return (uint32_t) *(env->memory_base) + guestOffset;
  } else {
    // invalid inputs or some part of buffer is OOB; trap
    while(1);
  }
}
```

Figure 4-4: Since the host environment and modules have different memory spaces, Plat translates between physical memory addresses and module memory offsets in order to call functions. To ensure safety, these translation functions make use of the base address and length of the calling module that is included in the Z_env_instance_t struct. Every call from a module to an imported function passes an instance of this struct as the first function argument.

```
void init_usb_hid() {
  Z_usb_hid_init_module();

  Z_usb_hid_instantiate(&usb_instance, &usb_env);

  // make input/output buffers; save their location in physical memory
  inBuffer = translateGuestOffset(&usb_env,
    Z_usb_hidZ_allocate(&usb_instance, INOUT_BUF_SIZE), INOUT_BUF_SIZE);
  outBuffer = translateGuestOffset(&usb_env,
    Z_usb_hidZ_allocate(&usb_instance, INOUT_BUF_SIZE), INOUT_BUF_SIZE);
}

void usbhid_send(uint8_t * msg) {
  memcpy(inBuffer, msg, INOUT_BUF_SIZE);
  Z_usb_hidZ_usbhid_send(&usb_instance, translateHostAddr(&usb_env,
    inBuffer));
}
```

Figure 4-5: When calling a module function, such as usbhid_send, an interface layer copies data into a region allocated for each argument. After translating the region's address from host memory space to the guest memory space, the interface function passes the offset at which the data is located in module memory as a pointer argument to the exported function.

its own memory space as pointers. The host environment has access to module memory, so explicit copying of arguments from module memory into host memory is not necessary here, but care must be taken to validate the offsets provided by the module and to translate those offsets into physical memory addresses. As shown in Fig. 4-6, for example, the imported SHA256 function updates the current context with a given memory region. To allow this, the interface function first translates the provided guest offset to a host memory address then passes the offset and length to the underlying function without any copying.

```
/* import: 'env' 't_crypto_sha256_update' */
void Z_envZ_t_crypto_sha256_update(struct Z_env_instance_t* env, u32
    dGuestOffset, u32 len) {
    uint8_t* data = (uint8_t*)translateGuestOffset(env, dGuestOffset, len);
    return crypto_sha256_update(data, len);
}
```

Figure 4-6: When a module calls an imported function with a pointer argument, the interface layer checks that the entire memory range that the host function may access is within the bounds of module memory. This verification is performed by the function that translates the module-provided offset to a physical memory address: the translation will trap if the range is invalid.

Importantly, the arguments provided by the module are *untrusted*. A bug or vulnerability within the module may cause these to be any value, and we cannot trust that the module is providing valid offsets into its own memory. And since these offsets will be used by the host to determine which physical memory addresses it can access, missing or incorrect checks can compromise the integrity of the sandbox. If, when calling the SHA256 function from Fig. 4-6, the module provided arguments for `dGuestOffset` or `len` that cause any part of the source region to be outside of the bounds of module memory, the host would read potentially sensitive data outside of the module and provide it to the module as part of the hashed result. If instead of SHA256 the imported function were something like `memcpy`, the module could use this to read and write arbitrary memory locations and escape the sandbox.

To prevent defining imported functions that allow this kind of sandbox escape, `translateGuestOffset` verifies that the provided offset and length of memory to be read is within module boundaries before converting the offset to a physical memory location. If any part of the memory region to be converted is out of bounds, the translation function will trap and prevent a sandbox escape; otherwise, the function will convert the offset to a physical memory address and return the result.

To minimize the opportunity for bugs, all definitions of imported functions that receive a pointer argument from the module use `translateGuestOffset` to check that the memory region to be accessed by the host is within module memory and, if so, translate the offset to a physical address. Instead of many checks distributed throughout the many definitions of imported functions, using one common function for translation means fewer chances for bugs and a single piece of code to carefully reason about to achieve correctness.

This approach allows passing data structures of arbitrary length across the module boundary. It does not handle all use cases—for example, complex data structures with internal pointers would be difficult to handle with this approach due to the need to copy and translate pointers inside the data structure. Past work, such as RLBox's automatic pointer swizzling [30], handles this much more comprehensively at the cost of additional complexity. Rather than supporting the transport of complex data structures across the module boundary, we choose boundaries for privilege separation that do not require this kind of complex function argument, making our simple approach sufficient for our needs.

4.4 Memory Management

WebAssembly is not designed for embedded devices, but its design as a universal compilation target works quite well for most uses. It becomes clear that it was not meant to target resource-constrained systems, however, when trying to fit a WebAssembly module's memory in a small embedded RAM. WebAssembly's module memory is allocated in terms of *pages*, chunks of memory that are defined in the WebAssembly spec to be 64K bytes long. This means that no matter how simple the code inside a module, the current WebAssembly spec requires that module to be allocated at minimum 64K bytes of memory. This is convenient for runtime developers working on desktop CPUs, as it allows them to align module memory with page boundaries and use page table hardware to trap on out-of-bounds memory accesses.

However, most microcontrollers have very little RAM available and certainly do not have dedicated page table hardware. The nRF52840 chip used for our initial testing, quite a high-end chip, has 256K [31] of RAM. Most common embedded CPUs have far less: the STM32L432 chip used in the Solokey and that we use for Plat itself, for example, has only 64K [45] of SRAM available. If any other data is stored in memory at all, this is not enough memory for even a single WebAssembly page. To make it practical to run multiple modules simultaneously on an embedded platform, Plat must deviate slightly from the WebAssembly specification.

Prior work such as eWasm [32] has shrunk the granularity at which memory can be allocated in order to make it practical to fit modules in an embedded context. We follow suit, with some modifications to fit the off-the-shelf `wasm2c` compiler we are using. `wasm2c` supports modifying the WebAssembly page size easily. In our initial single-module tests, the modules we tested used less than 64K of memory and thus (after slight tweaks to the compiler settings to use memory more tightly) used a single page of memory. To make these single modules fit, we simply shrunk the page size from 64K to 32K—still enough to fit all memory that the module needed, but small enough to leave plenty of memory to use in the trusted code. Though the C-to-Wasm compiler is not aware of this change, it does not allocate anything in the upper half of the allocated 64K and thus the generated code never makes any memory accesses there. Since, like most code targeting embedded devices, the sandboxed C code does not perform any dynamic memory allocation, it is safe to check the memory usage at compile term and trim the module memory to remove the unused portion of the 64K page.

This approach does not extend beyond a single module. Since memory demands of each module are different, setting the page size in this way means that each module must be allocated enough memory to fit the demands of the larger module. If the two modules required 33K and 10K, for example, they should be able to fit in 64K of RAM. But by having each take a single page and modifying the page size globally, each module must be allocated at least 33K—too much to fit in a 64K RAM.

Instead, Plat's toolchain shrinks the runtime page size all the way to 1024 bytes. Since a single 1024B page is no longer be large enough to store an entire module's data, each module needs additional pages allocated initially to fit all its data. But the number of pages allocated to the module initially is defined by LLVM: after laying out objects according to its link order, LLVM divides the space needed by the WebAssembly page size to determine the necessary number of pages and includes that number in the binary, from where `wasm2c` reads it when determining how many pages to allocate initially. LLVM has the WebAssembly page size hard-coded into the source code: without recompiling LLVM, there is no way to tell LLVM that Plat uses a 1024B page size, and as such, LLVM-generated WebAssembly binaries by default list only a single page required for our modules that require several. Further, large amounts of static data in each module means that modules must have all their memory available when they are created: it would not be possible to have the module grow its memory after startup. Plat's toolchain works around this problem by using the `--initial-memory` flag to the Wasm LLD port [53]. After manually determining the desired number of initial pages n for a module, we pass the compiler $n \times 64K$ as the initial memory size. When LLVM performs its internal division by page size, this causes LLVM to include the desired number of pages in the WebAssembly binary it generates and thus causes `wasm2c` to allocate enough memory on module startup.

With this modification, we can control the size of each module's memory independently with 1K granularity, allowing us to tightly pack multiple modules into a small memory space with limited overhead.

5

Peripheral Driver Sandboxing

One crucial part of Plat's embedded privilege separation plan is to isolate the USB driver. Drivers have historically had many bugs, not only in embedded devices but also in systems like Linux [13, 47]. On embedded systems similar to a security key, the USB driver and surrounding code in particular has been a source of bugs [36, 37], so we sought to limit the damage these bugs can cause by sandboxing the entire USB stack. This way, bugs in the USB driver could corrupt the USB driver and even cause arbitrary code execution inside the sandbox, but by the guarantees of WebAssembly could not corrupt important memory outside of the driver or read sensitive data like private keys from outside of module memory. Plat provides special support for sandboxing drivers.

Drivers typically interact with device peripherals using memory-mapped IO: the device has a set of registers that are mapped into physical memory at specific addresses, and the driver reads and writes these registers to configure and control the peripheral. From within a WebAssembly module, of course, this is impossible: WebAssembly allows access to only a small region of module memory that is abstracted away from the underlying physical memory. While WebAssembly supports loads and stores to memory, all memory locations must be relative to and within this module memory. There is no way for a WebAssembly module to specify that it would like to read or write a specific physical memory location.

And indeed this is by design: most applications of WebAssembly do not require this direct physical memory access, and in systems with ASLR even allowing the module to learn a physical memory address would be a security issue [30]. However, sandboxing a driver certainly requires this level of access.

Plat implements a generally applicable *peripheral proxy* to allow drivers to be contained in a WebAssembly module. This peripheral proxy along with interrupt handling comprise most of what is needed to sandbox a USB stack for Plat's STM32L432 chip.

5.1 STM32L432 USB Peripheral

The USB peripheral hardware on the STM32L432 chip that Plat uses proved convenient for privilege separation. With the data transfer rates that USB enables, direct memory access, or DMA, is virtually required in a USB peripheral. DMA allows a CPU peripheral to read and write memory directly, avoiding the need to read data byte-by-byte from a peripheral's registers. In many microcontrollers this could have made our approach difficult to adapt, as DMA hardware would often be able to violate our sandboxing restrictions by writing anywhere in global memory. This would have been especially difficult if the peripheral's data structures involved complex in-memory pointers that are followed by DMA hardware as it reads from memory, as is common in packet-based peripherals like networking and USB. DMA hardware is able to access any physical memory location, and will access memory based on its configuration registers and the data structures it supports. If the data structures are simple, it is straightforward to ensure that a driver will not cause the DMA hardware to access sensitive data. But as data structures become more complex and involve more layers of indirection, it becomes more difficult to reason about how each of the driver's writes to memory will influence the DMA hardware's behavior. DMA is discussed further in Section 5.3.

The STM32L432's USB peripheral indeed uses a form of DMA, and uses exactly these in-memory pointers to point to packet memory, but it does so in a way that is much easier to reason about: the USB peripheral has a dedicated memory region, separate from the main RAM, that is accessible by both the main CPU and by the USB peripheral. This memory region is mapped in the peripheral address space, and the USB peripheral cannot access any memory outside of this USB memory. USB packets must be written to and read from this region in order for them to be sent: it is not possible to instruct the USB peripheral to read a packet from an arbitrary memory location in RAM. Thanks to this design, we avoid reasoning about values written to memory and instead can achieve functionality and security by reasoning only about the addresses the driver accesses.

5.2 Peripheral Proxy

To allow the USB driver to access peripheral MMIO, Plat's *peripheral proxy* implements a set of functions that allow reading or writing a small set of physical memory locations that correspond to certain device peripherals. Unlike the memory accesses supported by WebAssembly natively, these are not be indexed from the start of WebAssembly memory; instead, this imported function takes physical memory addresses. This is powerful, but gives an obvious vector to circumvent WebAssembly's guarantees: if naively implemented, at attacker with control over the driver module could use these functions to hijack control flow outside the module or read the secret key. In order to preserve isolation, these memory accesses are carefully checked to ensure that they read and write only from and to peripheral MMIO locations. If, due to a bug or other vulnerability, the driver tries to access other memory, the address checking code will trap and stop any further exploitation.

As shown in Fig. 5-1, the definitions of these functions are quite simple: they take in a physical address and optionally a value to write, check that the address is valid for the calling module's driver type (stored in `env`), and read or write the value if permitted. This simplicity is very important, since a bug in these memory access functions could compromise the sandboxing of the modules.

The address checking functionality is of crucial importance. Since bugs in the address checking code can similarly compromise the integrity of the sandbox, it is very important that the address checking code does not allow access to any addresses outside of the necessary peripheral MMIO. To reduce implementation effort and minimize the possibility of bugs in conditionals, we generate the checking code from a simple YAML specification, as seen in Fig. 5-2, which defines accessible peripherals for the two modules in Plat. Memory regions can be read directly from the processor datasheet and entered in to the YAML file with minimal opportunity for error. Though the range checks we generate using this specification are simple, programmatically generating them avoids possible errors in comparison and logic and enables easy modifications.

Fig. 5-2 shows that the USB HID module is allowed access to the USB, RCC, and PWR peripherals. The USB entry includes both the USB module's configuration registers and the dedicated USB memory region. RCC and PWR are peripherals that allow control over clock control and power gating for all peripherals, and are necessary for the USB driver to configure itself. While access to RCC and PWR does allow the USB HID module some access beyond that strictly necessary to perform

```
void mem_trap() {
  while (1);
}

uint32_t Z_envZ_memReadWord(Z_env_instance_t* env, uint32_t addr) {
  if (checkAddr(env, addr, false, 0))
    return (uint32_t) *(uint32_t *) addr;
  mem_trap();
  return 0;
}

uint32_t Z_envZ_memWriteWord(Z_env_instance_t* env, uint32_t addr,
    uint32_t value) {
  if (checkAddr(env, addr, true, value)) {
    *((uint32_t *) addr) = value;
    return 0;
  }
  mem_trap();
  return 0;
}

...
```

Figure 5-1: Full-word (32 bit) versions of proxy functions that allow safety-checked access to physical memory addresses. Parallel functions exist for half-word and byte granularities. The `checkAddr` function, from Fig. 5-2, limits access to only explicitly allowed peripherals.

its function, we do not anticipate this to be a security risk—though admittedly it is difficult to reason about and a more complete solution would be best. A slight modification to the `checkAddr` infrastructure to support specification of bit-level access to these registers would allow limiting the module's access to exactly the bitfields corresponding to the USB driver and would remove this risk.

This infrastructure allows memory accesses from a sandboxed module, but not via the standard WebAssembly load/store instructions which operate relative to module memory. Instead, we need to modify driver code to perform peripheral memory accesses via the `memReadXXX` and `memWriteXXX` functions shown in Fig. 5-1. While this modification can be tedious, it is the only modification necessary to allow a device driver to run inside a module. For instance, the line `USBx->CNTR = 0`, which clears the USB control register, becomes `memReadWord(&USBx->CNTR, 0)` when sandboxing the USB driver. We use a set of convenience macros, discussed further in Section 8.2, to simplify this process.

```yaml
device-classes:
  - name: USB
    allowed-regions:
    - start: 0x4000_6800
      end: 0x4000_6BFF
    - start: 0x4000_6C00
      end: 0x4000_6FFF
  - name: RCC
    allowed-regions:
    - start: 0x4002_1000
      end: 0x4002_13FF
  - name: PWR
    allowed-regions:
    - start: 0x4000_7000
      end: 0x4000_73FF

modules:
  - name: USBHID
    allowed-classes:
    - USB
    - RCC
    - PWR
  - name: FIDO2
    allowed-classes: []
```

device.yaml (source)

```c
bool checkAddr(struct
    Z_env_instance_t * env, uint32_t
     addr, bool write, uint32_t
    write_val) {

  switch (env->module_class) {
    case MODULE_CLASS_USBHID:
    return (
      // USB
      (addr >= 0x40006800 && addr <
    0x40006bff) ||
      (addr >= 0x40006c00 && addr <
    0x40006fff) ||
      // RCC
      (addr >= 0x40021000 && addr <
    0x400213ff) ||
      // PWR
      (addr >= 0x40007000 && addr <
    0x400073ff)
    );
    break;
    case MODULE_CLASS_FIDO2:
    return (
      false
    );
    break;
    default:
    return false;
    break;
  }
}
```

checkAddr.c (generated)

Figure 5-2: YAML-formatted input (left), seen here for our FIDO2 security key, specifies peripheral memory regions and permits individual modules to access those regions. This input is used to generate a series of checks (right) to determine if a given memory access is permissible. For non-driver modules such as FIDO2, we allow access to no drivers, prohibiting all memory accesses via the peripheral proxy.

5.2.1 Interrupt Vectoring

It is common for real-time device drivers to use interrupts to promptly handle various events, and the USB driver for the STM32L432 is no different. The USB stack uses an interrupt to read packets from the USB memory region as they are received. In the original Solokey code, the USB stack implemented a function to be called whenever a USB interrupt occurs.

To support this in the module context, Plat's USB WebAssembly module exports the interrupt handler, and the module interface code defines the necessary symbol to vector the USB interrupt to the exported module function, as shown in Fig. 5-3.

To configure and manage its interrupt, the USB stack requires access to the ARM core MMIO via the ARM CMSIS library. Instead of providing the USB HID module access to the entire range of core MMIO via the peripheral proxy, which would give the USB module significant unnecessary and possibly dangerous privileges, we provide the CMSIS interrupt management functions that the module needs as imports and check that the module configures only USB interrupts in the interface function. The module can use these functions exactly as if the module has native access to the core MMIO, but cannot access other ARM core functionality that it could have had we provided direct access to the core MMIO.

```
void USB_IRQHandler() {
  Z_usb_hidZ_USB_IRQHandler(&usb_instance);
}
```

Figure 5-3: The hardware USB interrupt vector points to a wrapper around the exported interrupt handler.

5.3 DMA Support

The approach discussed so far handles only peripherals with dedicated memory regions through which all IO happens. This is sufficient for Plat itself due to the STM32L432's USB Peripheral design, which uses dual-port memory for packet buffers instead of DMA, but in common use on many microcontrollers many peripherals are controlled via direct memory access (DMA): after some initial configuration via special registers, the peripheral reads directly from a region of memory without intervention, allowing the main CPU to dedicate resources to other tasks.

Before implementing Plat, our initial work was to explore the feasibility of sandboxing a device driver inside WebAssembly. Using an nRF52840 development board, which contains a powerful Cortex M4 chip and many peripherals, we implemented a sandboxed UART driver for two UART peripherals: standard UART, which supports a streaming interface where the application feeds a single byte at a time into the input register, and UARTE, which supports a DMA interface. Thus, this thesis includes a plan for handling DMA peripherals for systems like the nRF's EasyDMA.

Importantly, DMA adds additional complexity to the peripheral proxy's address-checking. Consider, for example, the UART with EasyDMA (UARTE) peripheral on the nRF52840 chip [31]. This peripheral includes `TXD.PTR` and `TXD.MAXCNT` registers for transmission and equivalent registers for receiving. After configuring `TXD.PTR` with a memory address, `TXD.MAXCNT` with a number of bytes to send, and finally triggering the `STARTTX` task via another register, the UARTE peripheral will act fully on its own. It will, independently from the CPU, read from memory starting at the address in `TXD.PTR` and send each byte over the wire.

A driver for this UARTE peripheral *must* write to `TXD.PTR` and `TXD.MAXCNT`—it is the only interface to the peripheral—but these registers can cause the peripheral to transmit data from anywhere in memory, potentially allowing a sandbox escape. In order to preserve our module isolation and ensure that the driver can access or send only data that was explicitly copied into module memory, more complex reasoning is required. Instead of inspecting only the address of a memory write as we were able to for non-DMA peripherals, we now need to create some model of the peripheral and consider the addresses that the (`TXD.PTR`, `TXD.MAXCNT`) pair allow the peripheral to access via DMA.

We implemented such a device model for EasyDMA on the nRF52840, which powers most of the chip's complex peripherals from UARTE to USB and even the 2.4GHz radio. All use a similar pointer-plus-length set of inputs, which is very convenient for this kind of checking. After modifying the specification format to allow specifying EasyDMA pointer/length register pairs, as seen in Fig. 5-4 for nRF52840's UARTE peripheral, we extended the check-generation code to consider the value being written as well as the address on writes. As shown in Fig. 5-5, the modified `checkAddr` code determines if the value to be written is part of a `PTR` or `MAXCNT` register and retrieves the current value of the other half of the pair. If the new pointer-plus-length pair would allow the peripheral's DMA to access memory beyond sandbox bounds, the checking code will disallow the write, preventing any change to the peripheral's behavior. Since reads to these registers cannot change

```
device-classes:
  - name: UARTE
    allowed-regions:
    - start: 0x40002000
      end: 0x40002570
    dma-registers:
    - name: rxd
      base: 0x40002534
      max-length: 0x40002538
    - name: txd
      base: 0x40002544
      max-length: 0x40002548
```

Figure 5-4: An extended version of the check-generation infrastructure supports specifying EasyDMA pointer/length register pairs. These pairs are used to generate address-checking code that considers the value being written as well as the address itself in order to ensure out-of-bounds memory accesses are disallowed.

the peripheral's behavior, there is no special checking for memory reads.

This approach works for the nRF52840's peripherals thanks to their easy-to-reason-about EasyDMA setup. However, not all peripherals on all chips are so simple. It is common to have much more complex configurations that, for example, have the pointer register point to a descriptor in memory that itself contains another pointer and so on, forming a linked list that is followed by the DMA hardware. For peripherals like this, a more involved model would be necessary, and past work [54] has explored checking driver behavior for complex peripherals.

Since the STM32L432 chip we used to implement the FIDO2 key uses a dedicated memory region shared between the USB peripheral and the application processor, even the simple DMA support that we implemented is unnecessary to sandbox the USB driver in Plat.

```
bool checkAddr(struct Z_env_instance_t * env, uint32_t addr, bool write, uint32_t
    write_val) {
  switch (env->device_class) {
    case DEVICE_CLASS_UARTE:
    if (write) {
      bool is_dma = true;
      uint32_t dma_reg_index = 0;
      uint32_t base_val;
      uint32_t length_val;
      switch (addr) {
        case 0x40002534:
          dma_reg_index = 0;
          base_val = write_val;
          length_val = *(uint32_t *) 0x40002538;
        break;
        case 0x40002538:
          dma_reg_index = 0;
          base_val = *(uint32_t *) 0x40002534;
          length_val = write_val;
        break;

        // ... another pair of base / length registers ...

        default:
        is_dma = false;
        break;
      }

      if (is_dma) {
        // check that DMA region base and length are within sandbox bounds
        uint32_t sandbox_start = (uint32_t) *env->memory_base;
        if (
          (base_val > (uint32_t) sandbox_start &&
          (base_val + length_val < sandbox_start + *env->memory_length))
        ) {
          return true;
        } else {
          return false;
        }
      }
    }
    return (
      // reads and non-dma writes use standard bounds checking
      (addr >= 0x40002000 && addr < 0x40002570)
    );
    break;
    default:
    return false;
    break;
  }
}
```

Figure 5-5: The extended check-generation infrastructure generates more sophisticated checks from EasyDMA register pairs and requires an additional input of the value to be written. Checking for memory reads are unchanged, but memory writes now have special handling to identify whether a base or length register is being written, read both new values of the modified pair, and ensure that the full DMA region is within module memory.

6

Secret State Protection

Plat's primary security goal is to protect the master secret from which all key data is derived. To achieve this, Plat splits the original Solokey FIDO2 library in two parts. Complex parsing and logic code is sandboxed in the FIDO2 module, and any cryptography code that touches the master secret or keys derived from it is run without sandboxing (Section 6.1). However, for the device to function through regular power resets caused by unplugging the security key, both the master secret and other non-sensitive data controlled by the FIDO2 module must be stored in persistent Flash memory. To ensure the protection of the master secret despite this shared storage, Plat implements a trusted *state manager* (Section 6.2) that safely controls access to persistent state.

6.1 Trusted Cryptography

The FIDO2 library necessarily performs quite a bit of cryptography. Not only does it need to sign the challenges that are the core of the WebAuthn protocol, but it also must hash the user's PIN, encrypt messages before sending them over USB, perform attestations, and more. In the Solokey codebase that Plat is built on top of, this cryptography code is a part of the library. However, in order to minimize the impact of possible bugs, our privilege separation goal requires ensuring that the module code cannot access the private key used for sensitive cryptography operations. Thus, we separate the cryptography code from the rest of the FIDO2 library code: while the majority of the library code is inside a WebAssembly module, the cryptography code runs natively as trusted code and the private key is stored outside of module memory, accessible only by trusted code.

This protects the secret key, but widens the boundary between the FIDO2 module and the trusted code: instead of needing only a few functions to enable hardware-specific behavior as imports, the FIDO2 module now needs to import every cryptography function that uses the secret key. This includes signature functions, but also includes functions for things like hashing and computing HMACs that support adding the secret key to the hash context.

However, not all cryptography computations need access to the public key. Some cryptography, such as simple hashing or performing an encryption key agreement with the host PC, requires only short-term secrets that we are not concerned about protecting. To minimize the size of the trusted code base and thus the exposure to bugs, we place only the cryptography code that requires access to the secret in the trusted environment. As much cryptography as possible is run inside the module, where the impact of bugs is still limited.

To enable this, we distinguish between standard cryptography functions, which use a `crypto_` prefix, and *trusted* cryptography functions, which use a `t_crypto_` prefix. Trusted cryptography functions are imported functions that are computed outside of the module and have access to the secret key. Non-trusted cryptography functions are computed inside the module, and do not have access to the secret key. If the module needs to compute a SHA256 hash of some unprotected data, it uses the in-module cryptography functions, but if it needs to compute a signature using the secret key, it must use the trusted cryptography functions.

In addition to protecting the master secret, separating these trusted cryptography functions from the majority of the FIDO2 module also allows Plat to enforce goals G3 and G4's restrictions on signatures: before Plat allows a module to request a signature, it ensures that the user has pressed the authenticator's button to approve the signature and increments the signature counter.

6.2 Persistent State Manager

Persistent storage in flash memory presents a particular challenge. In the Solokey code, all persistent state for the authenticator is stored in a single flash region. When the FIDO2 library wants to restore this state on startup, it calls a function `authenticator_read_state` that fills in an in-memory struct after reading the data from flash. The original FIDO2 library code then reads the master secret from this struct and passes it to the cryptography code for later use.

Our privilege separation plan, however, requires that the master secret never enters the FIDO2 module. To achieve this, we could either remove the master secret

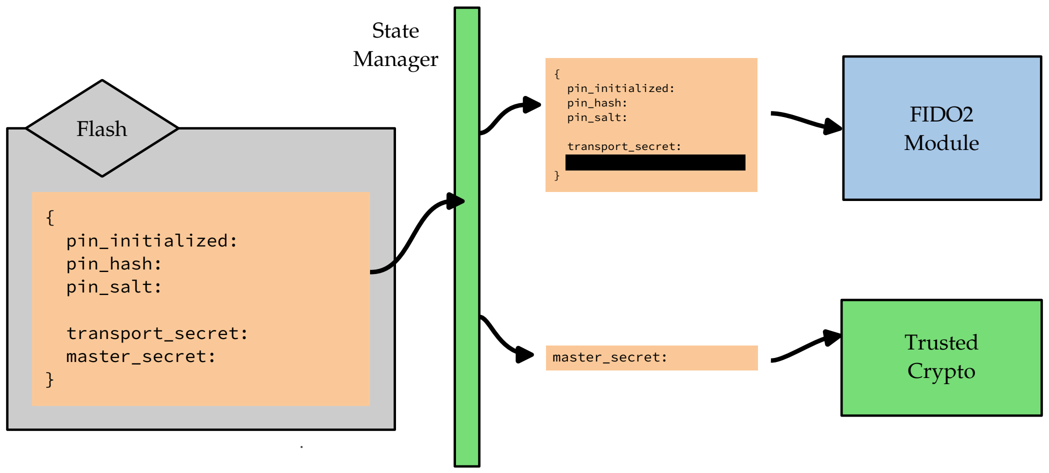

Figure 6-1: When the FIDO2 module requests to read the authenticator state from flash, the master secret used to generate the private key is first masked out of the state and loaded into the trusted cryptography module. This way, the FIDO2 module has access to sign data with the key but cannot access the key directly.

from the flash region accessed by `authenticator_read_state`, storing it elsewhere, or interpose on `authenticator_read_state` to remove the sensitive master secret from the data provided to the module. To minimize the need to modify existing code, we opt for the latter approach. As illustrated in Fig. 6-1, the interface function that provides the `authenticator_read_state` reads all data from flash, but before copying the data into module memory masks out the master secret, replacing it with zeros. The `authenticator_read_state` interface function passes the master secret to the trusted cryptography code without exposing it to the module.

Plat's state management system also provides the FIDO2 module with read-only access to the signature counter, preventing an attacker that compromises the module from modifying the signature counter.

7

Evaluation

We evaluate our privilege-separated FIDO2 token in terms of its security in the face of bugs as well and analyze several metrics about code size and performance. We compare Plat's performance on these benchmarks against the original SoloKey code, which we modify to remove its bootloader removed for simplicity and consistency with Plat. In Section 7.1 we measure the performance implications of our privilege separation approach and explore what it takes to get good performance. In Section 7.2 we measure the size of the trusted computing base with and without sandboxing enabled, and in Section 7.3 we compare the binary size of Plat and SoloKey. Section 7.4 shows the set of functions that each module imports and exports while Section 7.5 explores how this API and our privilege separation strategy would hold up in the face of several classes of bugs.

7.1 Performance

Privilege separation inherently adds overhead as it requires isolating trust domains from each other, and our approach of doing so with software fault isolation via WebAssembly includes additional overhead due to bounds checking and translation from WebAssembly's portable bytecode format. To quantify this overhead, we instrument the SoloKey and Plat FIDO2 module source code to time the individual steps of the registration and authentication interaction.

As seen in Table 7.1, while there was no significant difference in timing between SoloKey and Plat for the actual registration and authentication operations ("Make Credential" and "Get Assertion" respectively), the setup steps for these operations ("Get Key Agreement" and "Get Pin Token") both took over twice as long in the privilege-separated Plat than in the original SoloKey code.

Table 7.1: The timing of major operations in registration and authentication operations in native SoloKey and on the privilege-separated Plat key, with all optimizations enabled for each. Public-key cryptography operations are highlighted: in-sandbox operations are in yellow, while trusted operations that execute outside the sandbox are in pink.

	Time (ms)	
Step	**SoloKey (Native)**	**Plat (Sandboxed Modules)**
Get Authenticator Info	0	0
Get Key Agreement	218	542
Compute Public Key	207	530
Other	11	12
Get Pin Token	277	600
ECDH Shared Secret	208	529
Hash Shared Secret	0	0
Encrypt Pin Token	0	1
Other	69	71
Make Credential	382	398
Parse Request	1	3
Verify PIN Authenticity	1	1
Make Auth Data	215	221
HMAC Token	0	1
Calc. & Public Key	124	129
Other	91	92
Sign Auth Data	138	146
Attach Attestation	0	0
Other	30	27
Get Assertion	166	180
Make Auth Data	6	9
Sign Auth Data	138	142
Other	22	29

While the timing of all steps is accounted for primarily by the underlying cryptographic operations, two (in yellow) see a significant slowdown while two (in pink) do not. Importantly, the operations that did not see a slowdown are trusted cryptography operations: they involve the master secret, and thus require calls to functions that the module imports. The cryptographic computations themselves occur in the host, compiled directly from the source to the binary. The operations that did see a slowdown, on the other hand, such as the ECDH key agreement used to establish a key for symmetric encryption of the PIN between the host PC and the authenticator, do not require access to the master secret and thus are computed within the module without a call to the host. The source code for these cryptography operations is passed through the full C→ Wasm→ C→ asm toolchain.

7.1.1 Detailed Cryptography Performance Evaluation

In order to investigate the reason for this large difference in timing between sandboxed cryptography code and native cryptography code, we created a simplified system that contained two copies of the same cryptography code: one that we sandboxed using Plat's toolchain by compiling it from C to WebAssembly and back to C, and another running natively outside of a sandbox. In both cases, we use the same library implementation of the Elliptic-Curve Diffie-Hellman (ECDH) key agreement calculation, as this crypto code runs inside the sandbox in Plat and is a significant source of slowdown. With these two copies, we ran a series of experiments to determine the effects of compiler optimizations on the runtime of ECDH.

Table 7.2 shows the performance of the native and sandboxed cryptography respectively at various optimization levels. The native code is able to achieve its best performance through the use of hand-optimized assembly, enabled by setting `-DuECC_PLATFORM=5` at compile time. An inherent disadvantage of compiling to a portable format such as WebAssembly is the inability to include handwritten platform-specific optimizations, so we also compare the portable version of the cryptography code: that written in pure C without inline assembly. With the portable versions of each code, each with the maximum level of optimization supported by each, the Wasm-sandboxed version took about 2.5x as long as the non-sandboxed C version.

Using the maximally optimized WebAssembly code, the operation still executes quickly enough for the interactive use we require for Plat. However, this

Table 7.2: Performance of ECDH key agreement for sandboxed and native implementations. We compare a sandboxed ECDH implementation compiled using Plat's WebAssembly toolchain to both the C implementation and the hand-optimized ARM assembly implementation provided by the `uECC` library. The WebAssembly-sandboxed ECDH code achieves usable performance, but requires heavy optimization both of the generated WebAssembly binary and of the binary compiled form the `wasm2c`-generated code with GCC. The highlighted cells reflect the best performance we achieved with portable source code.

	GCC Opt. Level		
Source Type	-O0	-Os	-O3
`wasm2c` (Clang `-O0`)	40966	13121	4319
`wasm2c` (Clang `-Os`)	5697	1500	646
`wasm2c` (Clang `-O3`)	—	993	511
Native C	1410	312	210
ARM Assembly	197	139	125

requires trusting GCC's optimization passes to preserve correctness. For critical applications, this is not always a safe assumption: compiler bugs have been found in optimization passes [46]. Running GCC at `-O0` to avoid optimizer bugs that may introduce vulnerabilities, the best achievable performance for the hand-optimized assembly is hardly slower than the optimized version at 197ms, and even the native C code allows a workable 1410ms. Running the sandboxed ECDH operation, it is still safe to run Clang optimizations as the emitted code is still guaranteed to maintain WebAssembly's guarantees. But even with `-Os` in Clang, the best level of optimization that produces a binary small enough for our chip, the best achievable performance is a quite slow 5697ms. Good performance for `wasm2c`-emitted code simply requires an optimizing compiler. This reliance on GCC compiler optimizations for our `wasm2c`-based toolchain limits flexibility when preparing for possible compiler bugs.

In contrast to production WebAssembly runtimes built for browsers and servers which use page-table based techniques to implement bounds checks in hardware, our toolchain with `wasm2c` performs bounds checks naively with a simple conditional before each memory access. This is necessary on our embedded platform, which does not have hardware support for page tables, but adds significant overhead. In order to determine the source of the slowdown we saw, we modified the `wasm2c`-generated source for the cryptography module to skip these bounds checks, instead simply assuming that each passes. With bounds checks disabled, our fully optimized, Wasm-sandboxed ECDH operation took 295ms. This number is much closer to the 210ms elapsed during a non-sandboxed ECDH operation, showing that these bounds checks are responsible for much of the slowdown over native code.

Table 7.3: Comparison of ECDH compiled directly from C to ARM vs. ECDH compiled via Plat's sandboxing toolchain with and without bounds checking. Bounds checking is responsible for most of WebAssembly's slowdown.

Platform	**Bounds Checks**	**Time (ms)**
Native	N/A	210
Sandboxed	Enabled	511
Sandboxed	Disabled	295

Another necessary overhead of our current modularization toolchain is caused by the fact that module memory has its own address space. Because of this, every in-module memory access incurs a runtime translation to the corresponding physical memory location. This is likely responsible for additional overhead.

This analysis shows that while the performance of cryptography code sandboxed by Plat's toolchain is slower than natively compiled code, much of the slowdown comes from the bounds checks required by WebAssembly. WebAssembly's instruction format itself and the required transcompilation does not seem to be as significant of a factor as these bounds checks are.

7.2 TCB Size

In order to get an estimate of by how much Plat shrinks the size of the trusted computing base, we count lines of C code that are compiled with each approach. Table 7.4 shows that by moving the FIDO2 code and the USB stack code into privilege separated modules, and therefore out of the TCB, we reduce the number of lines of code that are ultimately trusted by over half. The FIDO2 module consists of 9025 lines of C code that are now isolated to their own sandbox, and the USB stack module contains 5843 lines of C code that are given access to only the USB peripheral. The sandboxing infrastructure, including runtime support for `wasm2c` and interface functions to module imports and exports, and some duplicated code in modules and in the trusted source, add additional lines of code to Plat that is not required in SoloKey, but most duplicated code is removed by the compiler. For example, the full cryptography library is included both inside and outside of

Table 7.4: Lines of C source code compiled in original SoloKey code and in Plat. Plat reduces the TCB size by over half, but has more total LOC than SoloKey due to sandboxing infrastructure and some duplicated code in and out of modules. This analysis does not include header files.

Codebase	**Sandboxed LOC**	**Trusted LOC**
SoloKey	0	19996
Plat	14868	8142
Trusted Crypto	0	2122
Startup & Libraries	0	4134
Wasm Runtime	0	299
FIDO2 Module	9025	1096
USB HID Module	5843	176

the module. Newly written code—the interface functions for module imports and exports—contributes around 319 lines of trusted code to Plat.

Though lines of code is not a perfect indication of bug exposure, this analysis demonstrates that Plat indeed has a significantly smaller trusted code base than the original SoloKey code. Certainly some of this code is more bug-prone than others, and we believe that we have sandboxed the most bug-prone code. But bugs are possible in any code, and thus this significant reduction in trusted lines of code is likely to correspond to a decrease in bugs in the trusted code base.

7.3 Total Code Size

As embedded systems are nearly always resource-constrained, the size of the compiled binary is of large importance. Our sandboxing approach has some inherent sources of overhead: for example, cryptography code that has trusted and untrusted versions, such as SHA256, is included both inside and outside of the module. Bounds checks must also be inserted at every memory access, increasing code size significantly: with range checks disabled for both modules, the same code with the same optimizations options shrunk by over 30%. Table 7.5 shows how privilege separation and our compilation process affect code size.

For embedded systems, which are typically dedicated to a single purpose, it is typically not absolute code size that matters. Rather, of primary importance is whether the binary will fit in the target device's flash memory. In our case, we were able to fit Plat on our target device while getting good performance, so did not seek

Table 7.5: Binary size of original SoloKey vs. Plat. There are a large set of possible optimization options to tweak; we present a set of options that achieves good performance and good code size by using `-Os` for most code and `-O3` for intensive cryptography code. In Plat, we use `-O3` for compiling the `wasm2c`-generated C code for the FIDO2 module that contains cryptography code and `-Os` Ellesmere.

Project	**Code Size (bytes)**
SoloKey (w/o Bootloader)	143820
Plat	215656
Plat (no module bounds checks)	164360

to reduce code size further. Finer-grained optimization controls could likely result in a smaller overall binary size for platforms that require it.

7.4 Module API

Each function that is exposed to the module is a hole in the module's sandbox and provides the module with additional permissions to perform actions that affect the host environment. Table 7.6 and Table 7.7 list the full set of imports and exports for the USB HID and FIDO2 modules respectively. Both modules achieve a simple interface and have access only to functions tightly related to their core purpose.

7.5 Bug Analysis

The advantage of Plat's privilege separation over the original monolithic SoloKey code is protection from bugs in untrustworthy parts of the system. To explore the effectiveness of this privilege separation, we consider several possible negative outcomes that could result from a bug and the security goals that prevent them:

- Secret Key Compromise: the worst-case scenario for a device like a security key is an attacker learning the secret key. This would allow the attacker to authenticate as the user any time they like. Goal G1, Secret Key Confidentiality, prevents this outcome.
- Permanent Token Corruption: If an attacker is able to modify the master secret, they will be able to permanently disable the token, locking the user out of their accounts. Goal G2, Secret Key Integrity, requires that an attacker is unable to modify the secret key.

Table 7.6: The full set of imported and exported functions for the USB HID module. The USB HID module allows quite a narrow interface, with only four exposed module functions plus the allocator. Beyond the peripheral proxy, it requires access only to safety-checked version of the CMSIS functions to configure its interrupts.

Imports	Exports
NVIC_EnableIRQ	usbhid_init
NVIC_GetPriorityGrouping	usbhid_recv
NVIC_SetPriority	usbhid_send
memReadHalfWord	USB_IRQHandler
memReadWord	allocate
memWriteHalfWord	
memWriteWord	

Table 7.7: The full set of imported and exported functions for the FIDO2 module. While the FIDO2 module exposes only a few functions to the environment, it requires an array of trusted cryptography functions as imports.

Imports	Exports
authenticator_read_state	allocate
authenticator_write_state	ctap_init
ctap_atomic_count	ctaphid_init
ctap_generate_rng	ctaphid_check_timeouts
ctap_user_presence_test	ctaphid_update_status
debugMarker	ctaphid_handle_packet
millis	
t_crypto_ecc256_derive_public_key	
t_crypto_ecc256_init	
t_crypto_ecc256_load_attestation_key	
t_crypto_ecc256_load_key	
t_crypto_ecc256_sign	
t_crypto_ed25519_derive_public_key	
t_crypto_ed25519_load_key	
t_crypto_ed25519_sign	
t_crypto_sha256_hmac_final	
t_crypto_sha256_hmac_init	

- Bypass User Verification: authenticators rely partly on their ability to guarantee user presence via a button press for security. If an attacker is able to bypass this verification and has control of the user's computer, they could authenticate as a user without their permission. Goal G3, User Signature Approval, requires that only one signature can be performed for a single button press, minimizing this risk.

- Corrupt Signature Counter: the FIDO2 specification includes a monotonically increasing signature counter that should be incremented with every authentication. An attacker that is able to decrease this signature counter could break the user's authenticator and an attacker who is able to artificially increase the counter could clone the authenticator. Goal G4, Signature Counter Integrity, prevents both of these possibilities by stating that the counter should be incremented only when a signature is approved.

Each of these outcomes are prevented by the security goals introduced in Section 1.3. We evaluate whether Plat achieves each of the security goals for several classes of bugs:

- USB stack buffer overflow: the USB code is complex and performs many copying and other buffer operations. A buffer overflow or memory read/write vulnerability in the USB stack could allow reading or writing unintended memory locations and even lead to remote code execution.

- CBOR parsing buffer overflow: the FIDO2 standard uses the Concise Binary Object Representation (CBOR) to encode its messages. If some of the encoding or decoding code contains a buffer overflow or other vulnerability such as the malformed struct in [39], adjacent memory could be leaked or modified. Similar bugs have been seen in the NibbleAndAHalf base64 parser used in several projects [35]. Buffer overflows could also lead to remote code execution.

- Stack overflow in FIDO2: a similar vulnerability as the stack overflow in [38] could lead to remote code execution.

- Non-constant-time cryptography: a cryptography implementation with error or oversight could reveal bits of the secret via timing channels.

- Incorrect cryptography use: faulty uses of cryptography can lead to, e.g., a padding oracle attack that reveals encrypted contents.

- RF side channels: an attacker with possession of the device may be able to reveal information about the private key via physical side channel analysis.
- Insufficient Hardware Randomness: as YubiKey experienced in 2019 [57], a hardware issue resulting in reduced randomness for key generation could make it possible for an attacker to guess secrets.

Table 7.8: Possible outcomes of various potential bugs for SoloKey and Plat. For each bug and security goal, a ■ indicates that both SoloKey and Plat maintain the security goal, a □ indicates that neither does, and a ▲ indicates that Plat maintains the security goal while SoloKey does not. There are no instances where SoloKey maintains the goal but Plat does not. While Plat protects the secret key and enforces button presses in the face of bugs within a module when SoloKey does not, neither protects against timing attacks, incorrect use of cryptography, or hardware vulnerabilities.

Bug	Secret Key Confidentiality	Secret Key Integrity	User Signature Verification	Signature Counter Integrity
USB Stack Buffer Overflow	▲	▲	▲	▲
CBOR Parsing Buffer Overflow	▲	▲	▲	▲
Packet Handling Stack Overflow	▲	▲	▲	▲
Non-constant-time Cryptography	□	■	■	■
Incorrect Crypto Usage	□	■	■	■
RF Side Channels	□	■	■	■
Insufficient Hardware Randomness	□	■	■	■

Table 7.8 shows that Plat offers effective defense against software vulnerabilities within a module: even in the face of buffer overflows potentially leading to arbitrary code execution or logic errors, Plat protects the secret key, enforces user presence for signatures, and ensures that the signature counter monotonically increases. Depending on the specific vulnerability, SoloKey's lack of privilege separation means that any of these vulnerabilities can compromise the whole system. For software bugs in the trusted host environment or for timing or hardware side channel vulnerabilities, however, neither Plat nor SoloKey offer protection. Finer-grained privilege

separation would help in decreasing the size of the TCB and thus of these unaddressed software bugs, but more sophisticated verification techniques and careful engineering are necessary to limit the likelihood of side channel vulnerabilities.

8

Discussion

We worked around several challenges when implementing Plat, from fitting WebAssembly modules onto embedded flash storage despite optimization for desktops to making necessary changes for sandboxing and finding bugs during our implementation.

8.1 WebAssembly Page Size

WebAssembly is convenient for use on embedded platforms due to its design as a universal compilation target. While the WebAssembly bytecode format is equally as well-suited for embedded processors as it is desktop ones, the fixed page size of 64Kb as set in the standard is a major problem for many embedded processors—especially when trying to run multiple WebAssembly modules alongside each other. This 64Kb is the least common multiple of minimum pages sizes on most desktop processors, and was set to enable page-table based bounds checking on MMU-enabled desktop machines. While this enables good bounds-checking performance for desktops, it is a showstopper for embedded devices—most of which barely have 64K of memory in total.

The workarounds discussed in Section 4.4 that Plat and similar work uses to avoid this incompatibility require little hacking of the WebAssembly runtime and compilation process. These small tweaks provide a 1Kb granularity for module memory size provide much more flexibility when using WebAssembly in the embedded realm, even allowing simultaneous execution of WebAssembly modules. A modification to the WebAssembly specification to allow smaller page sizes as an option—preserving a 64Kb size as the default for good desktop performance—would go far in enabling WebAssembly's use on embedded platforms.

8.2 Memory Access Replacement

Our current approach requires that sandboxed driver code treats accesses to peripheral memory entirely differently from accesses to data memory: instead of using simple C pointer dereferencing to read a driver memory location, for example, our approach requires that the driver calls a function like `memReadWord` with the desired address. The main challenge when sandboxing the USB HID stack, especially when compared to earlier tests of sandboxing UART drivers, was finding these accesses to peripheral memory and replacing them with calls to our memory access functions.

```
#define READ_WORD(REG) memReadWord(&REG)
#define WRITE_WORD(REG, VAL) memWriteWord((uint32_t*) &REG, (uint32_t)
    VAL)
#define READ_HWORD(REG) memReadHalfWord((uint16_t*) &REG)
#define WRITE_HWORD(REG, VAL) memWriteHalfWord((uint16_t*) &REG, (
    uint16_t) VAL)
#define READ_BYTE(REG) memReadByte(&REG)
#define WRITE_BYTE(REG, VAL) memWriteByte(&REG, (uint8_t) VAL)

#define ASSIGNAND_HWORD(REG, VAL) WRITE_HWORD(REG, READ_HWORD(REG)&VAL)
#define ASSIGNOR_HWORD(REG, VAL) WRITE_HWORD(REG, READ_HWORD(REG)|VAL)
```

Figure 8-1: Macros defined in USB HID module source code to simplify the process of replacing peripheral memory accesses with `memRead` and `memWrite` function calls.

A majority of the memory accesses were fairly simple to find: they were of the form `USBx->FIELD` or `hpcd->Instance->FIELD`. A basic find-and-replace regex combined with the convenience macros seen in Fig. 8-1 made replacing these memory accesses with function calls simple. But not all access to USB peripheral memory followed this convenient pattern: additional accesses were relative to a base address stored deep inside a struct or hidden away in a macro definition, making them difficult to find. Tracking down every memory access was a slow process of running the code until a trap due to an out-of-bounds memory access, tracking down the line of module C code that caused that out-of-bounds access, replacing the memory access in that line, and repeating several times.

More sophisticated analysis of the code than that which was possible with a simple regular expression could likely streamline this process. For example, Fig. 8-2 shows an instance of an access to peripheral memory that our regular-expression-based approach did not find and that had to be manually identified by tracing memory access traps. A static analysis approach with a model of C semantics, particularly one with support for taint tracking, could automate this replacement.

```
...
uint32_t BaseAddr = (uint32_t)USBx;
uint32_t i, temp;
uint16_t *pdwVal;
uint8_t *pBuf = pbUsrBuf;

pdwVal = (uint16_t *)(BaseAddr + 0x400U + ((uint32_t)wPMABufAddr *
    PMA_ACCESS));

for (i = n; i != 0U; i--)
{
  temp = *pdwVal;
  ...
  if (PMA_ACCESS > 1U)
  {
    pdwVal++;
  }
}
...
```

Figure 8-2: An instance of a memory access that our regular-expression-based replacement did not catch, in red. The chain of sources of the "tainted" pointer to the USB memory region are in purple. More sophisticated replacement that understands C semantics would make this replacement straightforward.

Once identified, the convenience macros from Fig. 8-1 make the replacement simple: wrapping `*pdwVal` into `READ_HWORD(*pdwVal)` solves the problem.

8.3 Debugging

A major pain point in Plat's implementation was our toolchain's lack of debugging support. Plat's toolchain involves several layers: first we compile from C to WebAssembly using `clang`, then from WebAssembly to C using `wasm2c`, and finally we compile the generated C to ARM assembly using `gcc`. The debugging support for the final C to ARM assembly step, of course, is quite robust: GCC generates debugging symbols that allow lining up individual ARM instructions with the source code that generated them. And debugging support in WebAssembly itself has been improving and has support for DWARF symbols as well [44], though only browser debuggers support these. However, the intermediate `wasm2c` step provides no debugging support at all, and when debugging the running code with `gdb` it is only possible to see the quite unreadable `wasm2c`-generated source.

To work around this, we inserted calls to an imported `debugMarker` function in the module source code. Even though it is a no-op, this imported function cannot be optimized out by clang since clang does not have knowledge of the imported function definitions. As shown in Fig. 8-3, this causes the calls to debugMarker to be emitted by wasm2c. Since the wasm2c output is available while debugging at runtime, instrumenting the code with these markers allows us to align the current position in the program with original C source code. This provides a method debugging even inside modules, where access to print statements may not be available.

```
...
w2c_i0 = w2c_l6;
w2c_i0 = w2c_add_cid(instance, w2c_i0);
w2c_B16:;
w2c_i0 = 10040u;
w2c_i0 = (*Z_envZ_debugMarker)(instance->Z_env_instance, w2c_i0);
w2c_i0 = w2c_p0;
w2c_i0 = i32_load(&instance->w2c_memory, (u64)(w2c_i0));
...
```

fido2_w2c.c

```
...
        add_cid(pkt->cid);
    }
}

debugMarker(10040);
if (cid_exists(pkt->cid))
{
...
```

ctaphid.c

Figure 8-3: Though the C code generated by wasm2c is essentially unreadable, inserting calls to an imported function (in red) provided enough cross-correlation between generated C code and source C code to enable debugging for Plat's implementation.

9

Conclusion

The Plat FIDO2 security key safeguards itself against bugs in its subsystems by isolating components like its USB driver and its parsers from core cryptography components that manage sensitive state. This thesis presents a toolchain and set of solutions for implementing this privilege separation for existing code on embedded systems, using WebAssembly as an intermediate format to provide safety guarantees. In addition to classical privilege separation, Plat's design includes solutions for challenges specific to the embedded environment, including delegating direct access to peripheral Memory-Mapped IO and managing persistent state that must be divided across trust domains.

9.1 Future Work

While Plat's toolchain is designed to be flexible, we have implemented only a single device—Plat—with the toolchain. Many of the included tools, such as the peripheral proxy infrastructure, could be extended to support broader use cases—and new tools altogether could make parts of the process much simpler. The implementation of additional privilege-separated devices using this framework would also certainly reveal further advantages and shortcomings of Plat's approach.

Though privilege separation is a powerful strategy, its use has been limited in deployed systems. Outside of the absolute most critical systems like web browsers, systems typically implement privilege separation only between large components such as individual services of a large company's web infrastructure. Good support does not currently exist for privilege-separating applications at the module or library level. Language-level support for privilege separation that removes the need for manual toolchain configuration and interface functions to link modules could increase the accessibility of privilege separation for small-scale systems.

Since Plat is based on existing software, its modules implement the API of the contained code. If designed from scratch, a modular approach to privilege separation could present other interesting possibilities. For instance, a common message-passing interface between all modules would enable the transparent insertion of a logging service between two modules and allow the system to guarantee that regardless of bugs in either module, malicious actions are recorded for later auditing.

Our analyses found that while there was significant slowdown in computation-heavy operations when sandboxed with WebAssembly, most of this overhead is caused by bounds checking. eWasm explored opportunities to minimize this overhead on existing hardware, but future explorations into modifications to embedded hardware to support fine-grained privilege separation could prove interesting.

9.2 Summary

Privilege separation has the capability to convert devastating bugs into trivial ones by means of “damage control”, but is difficult to implement effectively. Plat explores privilege separation in the new context of embedded devices, addressing the unique challenges that come with bare-metal execution and demonstrating that for many applications the overhead that comes with sandboxing is reasonable. We hope to see more systems adopt privilege separation as an additional layer of defense and hope for improved tools to make it easy to do so in the future.

Bibliography

[1] *1Password*. URL: `1password.com`.

[2] *32-bit ARM Cortex-M0 microcontroller; up to 32 kB flash; up to 10 kB SRAM and 4 kB EEPROM; USB device; USART*. LPC11U2x. Rev. 2.3. NXP Semiconductors. Mar. 2014. URL: `https://www.nxp.com/docs/en/data-sheet/LPC11U2X.pdf`.

[3] Josh Aas et al. "Let's Encrypt: an automated certificate authority to encrypt the entire web". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 2473–2487.

[4] FIDO Alliance. *FIDO Alliance - Open Authentication Standards More Secure than Passwords*. 2023. URL: `https://fidoalliance.org` (visited on 04/13/2023).

[5] Apple, Inc. *Apple, Google, and Microsoft commit to expanded support for FIDO standard to accelerate availability of passwordless sign-ins*. `https://www.apple.com/newsroom/2022/05/apple-google-and-microsoft-commit-to-expanded-support-for-fido-standard/`. 2023.

[6] Anish Athalye, M. Frans Kaashoek, and Nickolai Zeldovich. "Verifying Hardware Security Modules with Information-Preserving Refinement". In: *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22)*. July 2022.

[7] Anish Athalye et al. "Notary: A Device for Secure Transaction Approval". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP 2019)*. Hunstville, ON, Canada, Oct. 2019.

[8] Arnar Birgisson and Diana K Smetters. *So long passwords, thanks for all the phish*. 2023. URL: `https://security.googleblog.com/2023/05/so-long-passwords-thanks-for-all-phish.html` (visited on 05/04/2023).

[9] Joseph Bonneau. "The science of guessing: analyzing an anonymized corpus of 70 million passwords". In: *2012 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA, 2012. URL: `https://www.jbonneau.com/doc/B12-IEEESP-analyzing_70M_anonymized_passwords.pdf`.

[10] Joseph Bonneau et al. "The quest to replace passwords: A framework for comparative evaluation of web authentication schemes". In: *2012 IEEE symposium on security and privacy*. IEEE. 2012, pp. 553–567.

[11] Jay Bosamiya, Wen Shih Lim, and Bryan Parno. "Provably-Safe multilingual software sandboxing using WebAssembly". In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022, pp. 1975–1992.

[12] Haogang Chen et al. "Linux kernel vulnerabilities: State-of-the-art defenses and open problems". In: *Proceedings of 2nd ACM SIGOPS Asia-Pacific Workshop on Systems (APSys 2011)*. Shanghai, China, July 2011.

[13] Andy Chou et al. "An empirical study of operating systems errors". In: *Proceedings of the eighteenth ACM symposium on Operating systems principles* (2001).

[14] *Cortex-M0 Technical Reference Manual*. r0p0. Arm Limited. Mar. 2009. URL: `https://developer.arm.com/documentation/ddi0432/c`.

[15] Filippo Cremonese. *Security Analysis of the Solo Firmware*. `https://blog.doyensec.com/2020/02/19/solokeys-audit.html`. Feb. 2020.

[16] Fastly. *Compute@Edge*. 2022. URL: `https://docs.fastly.com/products/compute-at-edge` (visited on 01/09/2022).

[17] Dinei Florencio and Cormac Herley. "A large-scale study of web password habits". In: *Proceedings of the 16th international conference on World Wide Web*. 2007, pp. 657–666.

[18] Google. *OpenSK*. 2022. URL: `https://github.com/google/OpenSK` (visited on 12/23/2022).

[19] WebAssembly Community Group. *WebAssembly Design Documents*. 2022. URL: `https://github.com/WebAssembly/design` (visited on 04/13/2023).

[20] Andreas Haas et al. "Bringing the web up to speed with WebAssembly". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2017, pp. 185–200.

[21] HexView. *Inside Google Titan/Feitan Key*. URL: `http://www.hexview.com/~scl/titan/` (visited on 01/09/2022).

[22] HexView. *Inside Yubikey Neo*. URL: `http://www.hexview.com/~scl/neo/` (visited on 01/09/2022).

[23] Yongzhe Huang et al. "{KSplit}: Automating Device Driver Isolation". In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 2022, pp. 613–631.

[24] Maxwell Krohn. "Building Secure High-Performance Web Services with OKWS". In: *Proceedings of the 2004 USENIX Annual Technical Conference (USENIX '04)*. Boston, MA, June 2004, pp. 185–198.

[25] Ivan Krstić. *Behind the Scenes with iOS Security*. `https://www.blackhat.com/docs/us-16/materials/us-16-Krstic.pdf`. Aug. 2016.

[26] Philip Levis et al. "TinyOS: An operating system for sensor networks". In: *Ambient intelligence*. Springer, 2005, pp. 115–148.

[27] Amit Levy et al. “Multiprogramming a 64kb computer safely and efficiently”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 234–251.

[28] Robert Morris and Ken Thompson. “Password security: A case history”. In: *Communications of the ACM* 22.11 (1979), pp. 594–597.

[29] Shravan Narayan et al. “Going beyond the Limits of SFI: Flexible and Secure Hardware-Assisted In-Process Isolation with HFI”. In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 2023, pp. 266–281.

[30] Shravan Narayan et al. “Retrofitting fine grain isolation in the Firefox renderer”. In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 699–716.

[31] *nRF52840*. nRF52840. Nordic Semiconductor. 2023. URL: https://infocenter.nordicsemi.com/topic/struct_nrf52/struct/nrf52840.html.

[32] Gregor Peach et al. “eWASM: Practical software fault isolation for reliable embedded devices”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.11 (2020), pp. 3492–3505.

[33] proofpoint.com. *State of the Phish*. https://www.proofpoint.com/sites/default/files/threat-reports/pfpt-us-tr-state-of-the-phish-2023.pdf. 2023.

[34] Niels Provos, Markus Friedl, and Peter Honeyman. “Preventing privilege escalation”. In: *12th USENIX Security Symposium (USENIX Security 03)*. 2003.

[35] Christian Reitter. *Base64 parser issues in multiple projects*. 2020. URL: https://blog.inhq.net/posts/base64-parser-issues/ (visited on 12/20/2022).

[36] Christian Reitter. *KeepKey Key Erasure Vulnerability*. 2019. URL: https://blog.inhq.net/posts/keepkey-CVE-2019-18672/ (visited on 05/01/2023).

[37] Christian Reitter. *KeepKey receive buffer vulnerability (CVE-2019-18671)*. 2020. URL: https://blog.inhq.net/posts/keepkey-CVE-2019-18671/ (visited on 12/20/2022).

[38] Christian Reitter. *Trezor One dry-run recovery vulnerability*. 2019. URL: https://blog.inhq.net/posts/trezor-one-dry-run-recovery-stack-overflow/ (visited on 05/01/2022).

[39] Christian Reitter. *U2F init packet information leak*. 2019. URL: https://blog.inhq.net/posts/u2fhid_init_resp-information-leak/ (visited on 05/01/2023).

[40] Ledger SAS. *Supporting and Improving the Ledger Nano S*. 2019. URL: https://www.ledger.com/improving-and-supporting-the-ledger-nano-s (visited on 01/09/2023).

[41] *Secure authentication microcontroller*. A700x. Rev. 3.1. NXP Semiconductors. Mar. 2012. URL: https://www.mouser.com/datasheet/2/302/a700x_fam_sds-1187735.pdf.

[42] *Secure microcontroller with enhanced security and up to 320 Kbytes of Flash memory*. ST31H320. Rev. 1. STMicroelectronics. Nov. 2015. URL: https://www.st.com/resource/en/data_brief/st31h320.pdf.

[43] Solokeys. *solo1*. 2022. URL: https://github.com/solokeys/solo1 (visited on 04/13/2023).

[44] Ingvar Stepanyan. *Debugging WebAssembly with modern tools*. 2020. URL: https://developer.chrome.com/blog/wasm-debugging-2020/ (visited on 05/04/2023).

[45] *STM32L543KC*. DS11451. STMicroelectronics. 2018. URL: https://www.st.com/resource/en/datasheet/stm32l432kc.pdf.

[46] Chengnian Sun et al. “Toward understanding compiler bugs in GCC and LLVM”. In: *Proceedings of the 25th international symposium on software testing and analysis*. 2016, pp. 294–305.

[47] Michael M. Swift et al. “Recovering device drivers”. In: *ACM Trans. Comput. Syst.* 24 (2004), pp. 333–360.

[48] TheCharlatan. *List of Hardware Wallet Hacks*. 2019. URL: https://thecharlatan.ch/List-Of-Hardware-Wallet-Hacks/ (visited on 05/02/2023).

[49] Kenton Varda. *WebAssembly on Cloudflare Workers*. 2018. URL: https://blog.cloudflare.com/webassembly-on-cloudflare-workers/ (visited on 05/08/2023).

[50] Evan Wallace. *WebAssembly cut Figma’s load time by 3x*. 2017. URL: https://www.figma.com/blog/webassembly-cut-figmas-load-time-by-3x/ (visited on 05/08/2023).

[51] wasm3. *wasm3*. 2022. URL: https://github.com/wasm3/wasm3 (visited on 04/13/2023).

[52] WebAssembly. *WebAssembly Component Model*. 2023. URL: https://github.com/WebAssembly/component-model (visited on 04/13/2023).

[53] *WebAssembly lld port*. 2023. URL: https://lld.llvm.org/WebAssembly.html (visited on 04/13/2023).

[54] Dan Williams et al. “Device Driver Safety Through a Reference Validation Mechanism.” In: *OSDI*. Vol. 8. 2008, pp. 241–254.

[55] Bennet Yee et al. “Native client: A sandbox for portable, untrusted x86 native code”. In: *Communications of the ACM* 53.1 (2010), pp. 91–99.

[56] Yubico. *Press room*. 2022. URL: https://www.yubico.com/press/ (visited on 12/20/2022).

[57] Yubico. *Security advisory YSA-2019-02*. 2019. URL: `https://www.yubico.com/support/issue-rating-system/security-advisories/ysa-2019-02/` (visited on 05/01/2023).

[58] Wei Zhou et al. "Good motive but bad design: Why ARM MPU has become an outcast in embedded systems". In: *arXiv preprint arXiv:1908.03638* (2019).