Intra-Process Least Privilege and Isolation for Emerging Applications

Marcela Sofía Melara

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of Computer Science
Adviser: Michael J. Freedman

September 2019

Abstract

Third-party libraries reduce software development costs and effort. Designed for flexible reuse, libraries implement a small set of features, allowing developers to build applications by combining libraries that provide the desired functionality. However, third-party code also poses a great risk: because the source code is rarely inspected or even accessible by the application developer, bugs or vulnerabilities that can leak sensitive data may go unnoticed.

Yet, existing data protection tools are insufficient because they do not enforce least privilege, restricting each library's access to only those data it needs for its functionality. Prior academic proposals have addressed this issue with two main approaches: (1) running application components in separate processes for strong isolation, or (2) tracking individual data objects throughout the application to prevent unprivileged components from disclosing sensitive information. However, these approaches see limited real-world adoption because they introduce significant development overhead and integration complexity.

This dissertation proposes **intra-process least privilege**, a design principle that facilitates enforcing least privilege for application developers by restricting access at the granularity of individual library functions, and strongly isolating data within a single process address space.

We first present Pyronia, a privilege separation system for language runtimes that targets IoT device applications. To protect sensitive OS resources, Pyronia combines three access control techniques: system call interposition, stack inspection, and page table replication. Developers then specify data access rules only for directly imported third- party functions in a central policy.

We next present Griffin, a memory access control system for Intel SGX cloud applications. Intel SGX enables developers to run sensitive code inside an enclave, a hardware-protected memory region within an applications address space. However, in practice, developers often include untrusted third-party libraries in the enclave, giving them unfettered access to all in-enclave data. Griffin leverages Memory Protection Keys (MPK) to partition an enclave and assign per-compartment access rules. Developers declare sensitive data objects and access privileges for in-enclave functions. Griffin then automatically confines these data objects in MPK compartments.

Pyronia and Griffin demonstrate the effectiveness of our intra-process least privilege approach in today's privacy-critical applications while easing integration efforts for developers.

Acknowledgements

I would first like to give many thanks to my advisor Mike Freedman. His guidance since my time as a Master's student helped cement how I approach security problems in real systems, and his frank feedback on my research projects taught me the importance of distilling broad issues down to the core problem that needs to be addressed. I'm also incredibly thankful for Mike's understanding and support of my personal life decisions, without which I quite possibly would have not been able to complete this work.

I had the great opportunity to work with Ed Felten as my first-year PhD advisor. I appreciate Ed's different perspectives on my research, and advice on graduate school more broadly. I also owe a lot to Mic Bowman, who became an unofficial second advisor during my final year. Our candid conversations and his support at a professional and personal level have helped me gain more confidence in my research.

While not part of my dissertation, I want to give many thanks to my co-authors and external collaborators on CONIKS. I would like to thank Aaron Blankstein for, especially early on in my graduate studies, being a supportive mentor who was always willing to take the time to help me understand new problems I faced, and lend an ear whenever I needed advice. It has also been a true pleasure working with Huy Vu Quoc, Ismail Khoffi, and Arlo Breault on the OSS side of CONIKS. Their willingness to question our solutions to problems, coupled with their attention to detail, helped get our system to where it is today, and have made me a better software engineer.

I am also grateful to all of my Hobart and William Smith Professors for realizing early on in my undergraduate years that I would embark on the PhD journey before I considered research as a viable career path. It is thanks to their unbounded encouragement and guidance that I have been able to get to where I am today. I remember all of our conversations about school and life very fondly.

For the first three years of my PhD, I was fortunate to be co-located with the Security & Privacy Group and CITP, which allowed me to interact with incredibly talented researchers and fellow grad students with diverse research backgrounds and interests, many of whom have become good personal friends as well. To my colleagues at Apple and Intel Labs, I extend many thanks for welcoming me into their communities and for helping shape my research interests. The summers I spent in the Bay Area and Portland were invaluable both professionally and personally.

Finally, I am lucky to have such an amazing partner in Wade, who has stood by my side during all of the ups and downs, and who completely understands the sacrifices I made in order to complete my PhD and become the researcher I am today. To my family, who has been rooting for me from day one, thank you all so so much for your love and support. And last, but absolutely not least, I thank all of my friends, far and near, for always finding ways to cheer me up and remind me that we're all in this together.

Support. The research in this dissertation was supported by National Science Foundation CNS Award #1111734, and by an internship at Intel Labs.

To W and R,
who make everything better.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

We live in a largely digital world: Software applications help us perform myriad every-day tasks, including highly privacy-sensitive tasks pertaining to our finances, personal health, and businesses. With the advent of the Internet of Things, on the one hand, users can now micromanage every aspect of themselves, their home, or their businesses with small embedded devices dedicated to continuously gathering data. Cloud computing, on the other hand, allows us to process and store large swaths of data at ever-growing rates, employing hardware with stronger security features such as trusted execution environments.

Most users remain agnostic to the complexity, diversity, and distributed nature, of the underlying software that makes up the applications they interact with. The vast majority of users put stake in the public trust and reputation of software vendors and service providers. As applications become more complex, programmers increasingly rely on existing *third-party* software libraries to build their applications, in order to reduce development time and costs.

However, the problem with today's library-centric software development practices is that library developers are often unknown to *application* developers. Library source code is often unavailable, or difficult to examine, and studies have shown that programmers rarely have the time, expertise, or even authority, to prioritize security and privacy in their development process [20, 6, 7]. Thus, developers cannot be expected to fully inspect the source code of every library they need for their application, meaning that any vulnerabilities or bugs in imported third-party software may remain undetected.

Indeed, software supply chain attacks are becoming increasingly common, as adversaries leverage the widespread use of open-source libraries to disseminate malicious code, or otherwise cause havoc. For instance, in October 2018 security researchers discovered malicious packages uploaded to the Python Package Index that leveraged common misspellings of popular Python packages with the intent to be accidentally downloaded, and gain remote access to victim machines, or even to steal Bitcoin [33]. Similarly, thousands of cloud services were affected by various bugs discovered in the OpenSSL library since 2014, most notably Heartbleed which allowed encryption key material as well as other sensitive application data such to be leaked [99].

Nevertheless, the convenience of third-party libraries makes them indispensable to today's software development practices. Developers rely on community trust, especially in open-source libraries, hoping that the third party whose code they import has done their due diligence to eliminate any security vulnerabilities, and has not published malicious code.

Myriad prior proposals have sought to restrict untrusted third-party code and enforce least privilege [109] with two main approaches. The first traditional approach seeks to partition a monolithic application into multiple processes in order to strongly isolate libraries and control their access to OS resources (e.g., [27, 129, 26, 139, 137, 115]). In contrast, the second traditional approach attaches policy labels to individual sensitive data objects, and tracks these objects as they propagate through the application. Language-level information flow control (IFC) systems (e.g., [140, 48, 120, 37, 93]), in particular, are capable of controlling access to sensitive data at granularities as fine as individual bytes.

However, these two approaches introduce a considerable amount of cost and complexity: Developers must manually refactor or annotate their source code. Additionally, process isolation requires expensive IPC between the isolated components, and propagating IFC labels at run time incurs significant memory and performance overheads. Nonetheless, even recent in-application privilege separation systems for emerging compute technologies still employ these two traditional techniques (e.g., [44, 62, 34]).

This dissertation addresses the urgent need for novel defenses against threats posed by third-party libraries imported into today's complex, data-driven applications. We develop **intra-process least privilege**, a design principle for privilege separation systems that significantly eases adoption by controlling access to sensitive data at the granularity of individual library functions within a single process address space, through a *central* policy.

One key observation in our approach is that application developers already have implicit *coarse-grained* notions about the expected behavior of the third-party code they import. Intra-process least privilege enables developers to make these coarse-grained data access policies explicit, while allowing these rules to be automatically enforced by a run-time access control monitor. This design thus obviates the need for prior unintuitive or error-prone application refactoring or data flow analysis.

We apply intra-process least privilege in two different emerging application domains: the Internet of Things (IoT) and trusted execution environments (TEEs). Using domain-specific access control techniques, we built two privilege separation systems that can transparently enforce application-specific data access policies, and protect against data leaks and corruption by third-party code, with little effort from the developer. The two systems we present tackle challenges specific to the IoT and trusted execution environments, but demonstrate the flexibility and effectiveness of intra-process privilege separation.

1.1 Security Concerns in Emerging Applications

The use of third-party libraries is common across all application domains. However, certain domain-specific features make the use of untrusted code in both the Internet of Things and trusted execution environments uniquely dangerous.

Internet of Things. The IoT encompasses small embedded sensing devices that continuously collect a specific type of data about their environment, and upload it to cloud servers for further processing and storage. For instance, in the consumer space, smart thermostats monitor a home's temperature, and send this data to a remote server that allows users to manage their home's energy consumption online. Wearables such as fitness trackers gather vitals data that is stored in the cloud to help users track their exercise habits over time.

Concerns about data security in the IoT have been mounting in the wake of private data leaks in safety-critical technologies (e.g., [50, 53, 143]), and large-scale malware attacks exploiting vulnerable devices [123, 45, 29]. Unlike traditional desktop and server settings, IoT devices are mostly single-purpose running a dedicated, single application. As a result, vulnerabilities in third-party libraries within a process pose a much bigger threat than on more traditional platforms, precisely because of IoT applications' capabilities to handle highly sensitive and personal information. Nonetheless, the few practical data protection mechanisms available today only secure data in-transit to cloud servers [11, 95], or prevent attackers from running unauthorized code on devices [110, 65]. These ad-hoc safeguards are not designed to prevent data leaks from malicious or vulnerable third-party code *imported* into an application.

To address this crucial issue, we present Pyronia, an intra-process privilege separation system for IoT applications written in high-level languages. IoT applications typically make heavy use of OS resources, primarily the file system, device drivers, and network sockets. For this reason, Pyronia combines three access control techniques to protect sensitive OS resources at the granularity of third-party library functions: system call interposition, call stack inspection, and page table replication.

Trusted Execution Environments. Modern cloud computing services such as Amazon AWS [8], Microsoft Azure [90], and IBM Cloud [63] provide scalable, on-demand compute infrastructure. As more computation is performed on remote cloud servers for increasingly sensitive tasks such as online banking and health data processing, rising concerns over security breaches (e.g., [98, 52, 78] have driven cloud services to provide stronger security features for their customers.

Trusted execution environments (TEEs) such as Intel SGX [68, 84], in particular, are seeing wider adoption by popular cloud services [83, 94]. TEEs provide hardware-isolated execution areas in memory, called *enclaves*. By running only the most trusted application components in the enclave, TEEs enable developers to minimize the trusted computing base (TCB) of their applications thereby protecting sensitive application data. However, porting existing applications to TEEs requires major refactoring efforts, as TEEs provide a restricted interface to standard OS features.

To ease development efforts, TEE application developers often choose to run their unmodified application in a library OS (libOS) container that provides a full OS interface within the enclave. Yet, this large-TCB approach now leaves sensitive in-enclave data exposed to potential bugs or vulnerabilities in third-party code imported into the application. Importantly, because the TEE libOS and the application run in the same enclave address space, even the libOS management data structures (e.g. file descriptor table) may be vulnerable to attack, where in traditional OSes these data structures are protected via privilege isolation.

We address these issues with Griffin, a privilege separation system for large-TCB TEE applications that partitions an enclave into tagged memory regions, and enforces per-region access rules at the granularity of individual in-enclave functions. Griffin is implemented on SGX using Memory Protection Keys (MPK) [70] for memory tagging.

1.2 Thesis Outline

The rest of this dissertation is organized as follows. In Chapter 2, we review the Principle of Least Privilege, and discuss how its application has evolved as new compute platforms emerge. We then define the properties of intra-process least privilege more rigorously. Chapter 3 describes Pyronia, our privilege separation system for the domain of IoT.[1] Next, Chapter 4 details the Griffin intra-process privilege separation system for TEEs.[2] We present directions for future work and conclude in Chapter 5.

[1]Joint work with Michael J. Freedman and David H. Liu. Parts of this work have been published [89], and presented at [5]: [88, 87].

[2]Joint work with Mic Bowman (Intel Labs) and Michael J. Freedman. Parts of this work have been published [86].

Chapter 2

Intra-Process Least Privilege

The system design principle of intra-process least privilege that we develop in this dissertation has its foundation in the more general Principle of Least Privilege introduced in 1974 by Jerome Saltzer and Michael Schroeder [109]. In this chapter, we examine the evolution of computing platforms, and discuss why the emerging computing systems of today necessitate us to revisit and redefine this core system design principle.

2.1 Least Privilege: A Historical Review

Early Timesharing Systems. With the introduction of timesharing in mainframes in the 1960s and 1970s, multiple users with different levels of authority were able to share compute resources at the same time. This now meant that computers needed a mechanism for enforcing different privilege levels at run time. Previously, mainframes only performed single-user batch processing, which did not need any kind of access control.

To address this new data security concern, Saltzer and Schroeder introduced the *Principle of Least Privilege* in their 1974 paper "The Protection of Information in Computer Systems" [109]:

> "Every program and every user of the system should operate using the least set of privileges necessary to complete the job. Primarily, this principle limits the damage that can result from an accident or error. It also reduces the number of potential interactions among privileged programs to the minimum for correct operation, so that unintentional, unwanted, or improper uses of privilege are less likely to occur."

Thus, to protect sensitive information that requires different access privileges, Saltzer and Schroeder proposed a general model: "[...] build an impenetrable wall around each object that warrants separate protection, construct a door in the wall through which access can be obtained, and post a guard at the door to control its use" [109]. To control access to these isolated compartments and enforce least privilege, this model additionally requires an authorization mechanism, i.e., a way to identify users and determine which users are authorized to have access to which compartments.

For their mechanism to be practical, Saltzer and Schroeder required that different *principals*, i.e., users or programs acting on their behalf, be able to share protected information while keeping the different principals isolated from each other. Thus, they proposed two main implementation models that allow controlled sharing of sensitive data: access control list systems, and capability systems, which differ in how and when users are authorized to access a protected object.

In an access control list system, the "guard" of an object maintains a list of authorized users, each with a unique identifier, and must check at each access request whether the principal's identifier is present on the access list for the object. In capability systems, on the other hand, the "guard" maintains a single identifier for the object, and a principal maintains a collection of unforgeable identifiers to each object she is authorized to access. While authorization in this implementation model is accomplished simply by verifying that the presented identifier match the requested object's identifier, capability systems often require additional mechanisms to prevent identifier forgery and control capability delegation [109].

While Saltzer and Schroeder focus on describing how these two implementation models may be used to protect memory segments on a machine, they extend their models to create *protected subsystems*, a "[...] collection of program and data segments that is 'encapsulated' so that other executing programs cannot read or write the program and data segments and cannot disrupt the intended operation of the component programs, but can invoke the programs by calling designated entry points" [109].

In other words, each protected subsystem executes within an isolated context, but may cooperate with other protected subsystems running on the same machine to complete a particular task. As multiple subsystems would run on the same processor in practice, a mechanism for switching between the different subsystems' contexts would be required [109].

Around this time, Denning proposed a second approach for protecting information in computer systems. In her 1976 paper "A Lattice Model for Secure Information Flow" [40], Denning argues that access control mechanisms are insufficient for ensuring that sensitive information is not leaked to unauthorized users, as these mechanisms do not "[...] tak[e] into account information flow paths implied by a given, outstanding collection of access rights." [40]. That is, once an authorized principal has gained access to a sensitive object, the principal does not have control over *how* this object is used by other principals on the same system.

To enable the secure flow of information between principals on a system, Denning assigned a *security class* to each sensitive data object and each principal, and a *flow model* that describes for each security class whether information of a given class may be passed to principals in another class. Security is then enforced by monitoring all information flows between every operation performed by a given principal, and preventing the flow of information if any data passing violates the flow model.

Thus, Saltzer and Schroeder's protected subsystems laid the groundwork for the process abstraction in modern OSes that relies on memory virtualization for isolation. Denning's model for secure information flow, on the other hand, would be the foundation for later information flow control systems.

Desktops and Personal Computing. The emergence of personal desktop and laptop computers between the 1970s and the 1990s introduced a more complex relationship between users, programs, and the data they process than had existed in prior timesharing systems.

In this multi-user multi-application setting, the Principle of Least Privilege needed to be applied to different users sharing a platform, as well as mutually distrusting applications running under the same user authority. To enforce least privilege between different users, PC operating systems most prominently provided discretionary access control mechanisms, restricting access to system resources based on the "owner" of a given resource and the permissions they have granted other users of the system. Additionally, OSes employed memory virtualization to isolate individual application processes run by the same user.

In contrast, mandatory access control (MAC) systems that enforce least privilege based on centralized *application-specific* access policies for OS resources, such as AppArmor and SELinux in Linux, were not introduced into desktop OSes until the late 1990s to early 2000s [13, 102]. Around this time, we also see proposals from academia for three main least privilege approaches: (1) confining untrusted application components in separate processes [49, 26, 30], (2) system call interposition [71, 24, 103, 46], and (3) decentralized information flow control mechanisms for tracking how data propagates at the level of programming language primitives [93, 92] as well as OS-level primitives [142, 76, 141, 42].

The Internet and Mobile Computing. The new millennium ushered in the popularization of the Internet, as well as increasingly capable mobile devices. By the 2010s, billions of people had access to an evergrowing selection of online services through mobile apps running on smartphones. However, with the ease of sharing data, including source code, on the Internet, and the higher demand for very rapid application deployment, the use of third-party software libraries in applications grew exponentially.

The collection of our personal data also increased very rapidly at this time, as mobile applications now facilitated highly sensitive personal tasks such as banking, private communications, and health management. To address the privacy concerns surrounding mobile apps that make heavy use of third-party code, a large body of research developed mobile-specific MAC systems [138, 25, 19, 31, 119], as well as privilege separation systems that isolate certain third-party libraries in separate processes [101, 115, 122, 144, 73, 61, 108, 58]. More recent proposals for Android have presented in-app privilege separation systems [113, 133, 18].

Computing Today: small IoT to large Cloud. Today's emerging platforms range from small, embedded IoT devices with limited CPUs and memory to powerful cloud servers capable of processing terabytes of data within short periods of time. Despite their differences in size and computational power, both platforms generate and process increasingly sensitive data. The trend towards library-centric application development also continues.

Least privilege requirements are vastly different between the IoT and cloud platforms. On the one hand, the majority of IoT devices are single-purpose running only a single dedicated application, with the primary threat stemming from untrusted third-party libraries. Privacy is indeed a major concern for many IoT application developers [23]. A small number of academic proposals for privilege separation in the IoT have presented systems that prevent sensitive data leaks by tracking how sensitive information flows between different components [44, 72], or by running different application components in separate processes [44].

However, state-of-the-art protections secure data in-transit to cloud platforms [11, 95], or limit remote code execution on individual devices [110, 65]. These ad-hoc safeguards do not prevent vulnerable and malicious third-party code within an application from leaking sensitive information.

Cloud servers, on the other hand, run in multi-user/multi-tenant settings, where the main security concerns stem from cross-tenant attacks and cloud platform compromise. To address these issues, server CPU manufactures have introduced new *hardware* technologies for enforcing stronger isolation between tenants and protecting sensitive application memory within the same address space in trusted execution environments (TEEs).

TEEs, such as Intel SGX [68, 84] or ARM TrustZone [16], have made it possible for developers to reduce the TCB of applications by placing only the most security-critical or privacy-sensitive functionality in the TEE, and leaving the majority of application functionality in the *untrusted* context.

However, TEEs typically require significant application refactoring efforts, which have led developers to incorporate untrusted third-party code, and even entire library OSes (e.g., [118, 34, 17]), into their TEEs, thereby increasing the TCB of their application. Much like in the IoT, developers' use of untrusted third-party code in TEEs has left sensitive applications vulnerable to bugs and exploits from within. Proposals to enforce least privilege within TEE applications rely on separating a monolithic application into multiple communicating TEE processes [34, 118, 62].

2.2 Motivation

Overall, we observe two main approaches to enforcing least privilege of intra-process principals. As new technologies have emerged, process isolation has been the primary approach to confine untrusted application components to their own address space. However, process isolation imposes significant development overheads: Developers may have difficulty cleanly separating components into processes and replacing inter-component interactions with expensive inter-process communication.

The second traditional approach for enforcing least privilege is information flow control (IFC), which attaches policy labels to individual sensitive data objects, and tracks these objects as they propagate through a system. Language-level IFC systems (e.g., [140, 48, 44, 120, 37, 93]) in particular, are capable of controlling access to sensitive data at granularities as fine as individual bytes. However, much like process isolation, IFC systems introduce a considerable amount of cost and complexity: Developers must typically manually annotate their source code to specify their object-specific policy labels around sensitive data sources and sinks.

These usability issues raise questions about the adequacy of these traditional least privilege approaches at a time when our applications handle highly sensitive data in the presence of potentially vulnerable or malicious code within an application. To address these problems, we propose **intra-process least privilege**, a novel approach for fine-grained access control that avoids any application refactoring or annotations.

2.3 Properties

Privilege separation systems that enforce intra-process least privilege have the following security properties:

P1: Function-level MAC. Access permissions to a given sensitive data object are enforced based on the programming language-level *function* requesting access. This property allows multiple intra-process functions with different privileges to operate on the same sensitive data object. Function access privileges are specified in a central application-specific policy.

P2: Least Privilege. A function may only access those sensitive data objects (e.g., OS resources, in-memory data) that this function needs to provide its *expected* functionality.

P3: Single Address Space Isolation. Sensitive in-memory data objects are confined in isolated least-privilege compartments *within* the application's address space.

2.4 Primitives

Intra-process privilege separation systems rely on four core primitives.

A ***sensitive data object*** represents an individual OS resource (e.g., file, device, socket) or unit of in-memory data (e.g., a cryptographic key) that the application developer wishes to protect. Each data object has an associated *label*, a human-readable string uniquely identifying an object in the policy and run-time access control monitor. For OS resources, the object label can be the identifier created by the OS; for files, for instance it may be the file path.

In the ***access policy***, the application developer specifies the sensitive data objects that individual language-level functions in their application may access, and the associated access privileges. Since developers already have an implicit understanding of the purpose of imported libraries, intra-process least privilege policy rules make explicit the developer's high-level descriptions of library functions' expected data access behaviors.

Memory domains represent isolated memory compartments that hold sensitive in-memory data objects within the application's address space.

Dynamic execution sandboxes create boundaries around a given function to execute it with temporarily elevated access privileges according to the developer's access policy. Upon returning from this function, the execution sandbox immediately revokes access to any sensitive data objects to limit any further access outside the scope of the function.

2.5 Summary

Saltzer and Schroeder laid the foundation for process isolation-based information protection in their 1974 paper. Denning first introduced a model for information flow control in computer systems in her 1976 paper. However, we find that existing privilege separation systems that rely on these two approaches are rarely adopted by developers in practice because they are difficult and inconvenient to use: these approaches lack an intuitive or familiar programming model, and they are often applicable only to a small subset of applications within a domain (e.g., [44, 27, 115, 120]).

In contrast, the only burden intra-process least privilege places on developers is to make explicit their *coarse-grained* expectations about the data access behavior of library functions they import. While this property of our approach does not protect data at the ultra-fine granularities of certain IFC systems, as developers' policies will inevitably omit dependency functions within the scope of their *authorized* functions, we believe that the benefits of providing an approach with a more intuitive programming model that is broadly applicable within an application domain greatly outweigh the loss of protection granularity.

Thus, intra-process least privilege achieves the goal to protect applications against sensitive data leaks in untrusted third-party code with two key primitives: memory domains, the "walls" around protected information, and dynamic execution sandboxes, which represent intra-process "protected subsystems" in Saltzer and Schroeder's model.

Memory domains enable us to partition a single process address space into protected compartments with isolation properties akin to those of traditional process isolation. To implement memory domains, we leveraging advances in software and hardware memory protection mechanisms that can be integrated into language runtimes (as in Pyronia §3) or library OSes (as in Griffin §4). Such memory domains then allow us to automatically partition an application's address space, and to enforce function-level data access policies on behalf of developers, without the tedious application refactoring required for traditional process isolation-based systems.

Dynamic execution sandboxes ensure that functions may only access those sensitive data objects authorized by the developer's data access policy. At each transition between authorized functions, intra-process privilege separation systems then adjust the application's access privileges to sensitive data objects based on the subsequent function in the application. In this manner, dynamic execution sandboxes preserve an application's original inter-component interactions, while enforcing least privilege of different functions. Thus, intra-process least privilege can also implicitly control how sensitive information flows between different principals, i.e., functions, without prior data flow analysis or manual code annotations for policy specification.

In the following chapters of this dissertation, we present two privilege separation systems that put in practice our intra-process least privilege design principles in the application domains of the Internet of Things (§3) and trusted execution environments (§4). Our two systems demonstrate how intra-process least privilege can be applied to enhance the security of applications in two different general domains, and greatly facilitate the process of enforcing least privilege for developers.

Chapter 3

Pyronia: Redesigning Least Privilege and Isolation for the Age of IoT

3.1 Introduction

Concerns about data security in the Internet of Things (IoT) have been mounting in the wake of private data leaks in safety-critical technologies (e.g., [50, 53, 143]) and large-scale malware attacks exploiting vulnerable devices [123, 45, 29]. These concerns have driven device manufacturers and application developers to deploy measures that secure data in-transit to cloud platforms [11, 95], or that prevent attackers from running unauthorized code on devices [110, 65]. Yet, these ad-hoc safeguards do not prevent vulnerable and malicious third-party code within an application from leaking sensitive information.

To address this crucial issue, we present Pyronia,[1] a privilege separation system that protects applications against exploits and vulnerabilities in untrusted third-party code with unmodified source code and minimal developer effort.

In Pyronia, we retain the goals of controlling how an application, specifically which *third-party library functions*, may obtain data from files and devices and to which remote network destinations an application may export data. By combining three *access control* techniques, Pyronia overcomes the limitations of prior approaches to achieve both fine-grained privilege separation and strong isolation without losing flexibility.

Our design hinges on one important observation: libraries are not monolithic. The median Python library used in IoT applications, for instance, has a dependency graph (i.e., number of nested library dependencies) 25 levels deep (see §3.2). Thus, while propagating data labels dynamically through the application in IFC-fashion based on a decentralized access policy would enable finer-grained data leak protection, reasoning a priori about all sensitive data paths within an application to prevent any data leaks proves to be a cumbersome and error-prone task.

In contrast, as an access control-based system, Pyronia enforces a *central* policy that specifies access rules for *directly imported* third-party library functions and the specific OS resources they may access. For example, to ensure that a sensitive image `face.jpg` can

[1]Pyronia being a gatekeeper for third-party libraries, is named after the genus of butterflies known as gatekeeper butterflies.

only be accessed by a third-party facial recognition library function *recognize_face()*, the developer specifies a read rule for the input `face.jpg` in her policy. Pyronia then blocks all attempts by any other library functions to access the image file. Thus, developers need not reason about third-party dependencies that may be unavailable or unfamilar to them, and application and library source code can remain unmodified.

To enforce such *function-granular* access control, Pyronia leverages the following three techniques.

1) **System call interposition** (§3.4.1) guarantees that access to OS resources by all application components can be controlled, even for native libraries integrated via a high-level native interface such as the Java JNI or the Python C API. However, system call interposition has traditionally only been realized as a process-level technique, and thus cannot handle intra-process access checks.
2) **Stack inspection** (§3.4.1) allows Pyronia to identify all library functions involved in the call chain that led to a given system call. Specifically, borrowing ideas from fine-grained privilege separation proposals in the mobile space such as [113] and [133], Pyronia checks the language runtime call stack to determine whether to grant access to a requested resource based on the full provenance of the intercepted system call.
3) **Page table replication** (§3.4.2) enables userspace processes to partition their own address space into isolated memory domains, and control access to specific memory regions. Pyronia uses this technique to prevent native code, which resides in the same address space as the language runtime, from tampering with the call stack in an attempt to bypass Pyronia's resource access control checks.

We implement Pyronia for Python and Linux (§3.5). However, we believe our approach can be applied to other language runtimes. Our prototype Pyronia system acts as a drop-in replacement for the CPython runtime, and includes a custom Linux kernel with support for page table replication. Our stack inspection-based system call interposition component is a kernel module built on top of the AppArmor [14] MAC system. To facilitate integration into a language runtime, we have implemented a Pyronia user-space library, which provides an API for automatically loading a developer-supplied resource access policy, as well as a memory domain management API.

We evaluated Pyronia's security and performance via three open-source IoT applications (§3.6 and §3.7). We demonstrate that Pyronia provides intra-process least privilege enforcement with moderate execution time and modest memory overheads. In particular, while the mean execution time overhead is not trivial, we find that it would remain imperceptible for IoT applications that only passively collect data without much user interaction. Microbenchmarks indicate that these overheads can be attributed mostly to the cost of dynamic access privilege adjustments to Pyronia's memory domains to enable Python's garbage collection. Maintaining the call stack memory domain requires only 0.33 kB of additional RAM per domain page (see §3.4.2), and a maximum of 28.1% additional memory for the entire application.

Table 3.1: Reported Python library vulnerabilities and number of unique libraries by attack class, from 2012-2019.

Attack class	**# Reports**	**# Libs**	**Lib/framework (# reports)**
Arbitrary code execution	28	24	python-gnupg (4)
MITM	19	14	urllib* (3)
Web attack	18	12	urllib* (4)
Denial of service	17	12	Django (3)
Direct data leak	12	10	requests (3)
Weak crypto	11	10	PyCrypto (2)
Authentication bypass	9	6	python-keystoneclient (3)
Symlink attack	6	4	PIL (2)
Replay/data spoofing	3	3	python-oauth2

3.2 IoT Application Development Today

Before designing Pyronia, we first sought to understand the landscape of IoT application development. To do so, we conducted an in-depth study of 85 open-source IoT applications and their libraries written in Python, as well as a brief analysis of reported vulnerable third-party Python libraries. Our analyses focus on Python because it is one of the most popular IoT development languages [23].

3.2.1 Library Vulnerabilities

Our survey of reported Python library vulnerabilities covers 123 reports created between January 2012 and March 2019 in the Common Vulnerabilities and Exposures (CVE) List [124]. We identify 78 distinct vulnerable Python libraries, and 9 main attack classes (see Table 3.1)[2].

We include shell injections under arbitrary code execution, and the majority of web attacks comprise cross-site scripting and CR/LF injection attacks. We classify vulnerabilities arising from accidental data exposure as direct data leaks. Authentication bypass vulnerabilities arise due to system misconfigurations or credential verification bugs. A symlink attack allows an adversary to gain unauthorized access to a resource via a specially crafted symbolic link.

While direct data leaks account for only about 10% of the reported vulnerabilities, we emphasize that the actual number of reported data leak vulnerabilities is significantly higher since most other attack classes, most notably arbitrary code execution, man-in-the-middle (MITM), and authentication bypass, lead to information leakage as well. Our analysis also demonstrates the diversity of vulnerable libraries, with a small number of libraries having a handful of reports in each attack class. The two Python packages with the most overall CVE reports in our survey are the widely used Django web framework and urllib* HTTP library family, each with eight reports. These findings underscore the degree to which IoT

[2]For a full list of the CVE reports included in our analysis, see Appendix A.

Table 3.2: Analysis of direct imports and dependency graph depth in a set of 85 Python IoT applications and in the top 50 third-party libraries imported by these applications. Unless noted otherwise, results of per-application analyses are shown.

	min	**median**	**mean**	**max**
# direct imports	1	8	13	253
# direct 3rd-party imports	0	3	6	186
# dependency graph depth	1	25	22	37
# per-lib dependency graph depth	0	27	24	34

Table 3.3: Five most popular Python IoT libraries, and the percentage of the 85 surveyed apps which included them.

Library	**Frequency**
RPi.GPIO	47.1%
requests	23.5%
picamera	21.2%
serial	12.9%
alsaaudio	11.8%

application developers are exposed to potential security and privacy threats by importing third-party code.

In addition, we find that creating a malicious library is fairly straightforward. In our lab setting, we built a Python module and a native library that exploit dynamic language features of Python, such as introspection, to replace function pointers at run time with malicious functions, leak data at import time, and are able to perform various modifications to the Python runtime's call stack. Importantly, these attacks are possible because Python exposes the exploited introspection APIs and features to provide programmers with more flexibility, visibility, and control over a module's functionality.

While we have not identified such attacks in the wild, our experiments demonstrate that these dynamic features and open APIs place too much trust in third-party library developers, and can be misused for nefarious purposes. Thus, both dynamic language features and the capabilities of native libraries pose threats to the integrity of the application itself and the privacy of user data.

3.2.2 Application Analysis

To better understand the impact of third-party libraries on how today's IoT applications are designed, we analyzed 85 open-source IoT applications written in Python. We obtained these applications primarily from popular open-source IoT development platforms such as `instructables.com` and `hackster.io`; our search focused on three broad categories of applications—visual, audio, and environment sensing—which we believe are representative of today's most privacy-sensitive IoT use cases. Our analysis answers four questions.

Table 3.4: Characteristics of the Top 50 Python IoT Libraries. These statistics include the 40% of the top 50 libraries that exhibit multiple of these characteristics.

Library feature	**% of top 50 libs**
Written in Python	12.0%
Have Native Deps	82.0%
Run external binaries	40.0%
Use ctypes	40.0%

1. To what extent are third-party libraries used in IoT applications? All but one of our sampled applications (98.8%) import at least one third-party library, with a mean of about 6 *direct* third-party imports per application. The maximum number of direct imports we found in a single application was 186 (see Table 3.2).[3]

2. How diverse is the third-party library landscape? Given that the Raspberry Pi single-chip computer is a very popular development platform for IoT, our sampled applications heavily target this platform. For instance, two of the top 3 most imported third-party libraries (`RPi.GPIO` and `picamera`) are Raspberry Pi-specific Python libraries (see Table 3.3). Nonetheless, overall we found 331 distinct third-party Python libraries among the 418 total imports in our sampled set of applications.

3. How heavily do IoT applications rely on native code? All of the third-party libraries in our study are Python libraries, i.e., they provide a Python API. However, because the CPython runtime provides seamless integration with native code, gaining insight into the prevalence of libraries with native dependencies would reveal how exposed the Python runtime is to threats in native code. Table 3.4 shows that among the top 50 third-party libraries in our study—based on the number of times they were included in our surveyed applications—only 12% are implemented purely in Python.

In contrast, 82% of the top 50 libraries include a component written in C/C++, among which we identified 68 distinct native dependencies. The remaining 6% of libraries only load a native shared library via the `ctypes` Python foreign function interface. Furthermore, about 46% of the sampled *applications*, and 40% of the top 50 third-party libraries, execute external native binaries (including a Python sub-process).

4. How deep is the dependency graph of the average IoT application? Finally, we analyzed the depth of the dependency graph (i.e., the longest chain of nested libraries) of each application and top-50 library in our sample, and use the depth as an indicator of application or library complexity. Across the 85 sampled applications, the median dependency graph depth is 25 levels, while the median library has a dependency graph 27 levels deep (see Table 3.2). These numbers highlight the great effort that would be required of developers to separate each library into its own process, or identify the sensitive data flows within a single application, to prevent all data leaks.

[3]It is worth noting that imported libraries are typically not inspected by package managers (e.g., PyPI) for security.

In summary, third-party libraries present a serious threat to sensitive IoT data. The single-purpose nature of most IoT devices means that attacks from imported libraries within an application's process are more realistic than those across process boundaries, as seen in traditional desktop and server settings. A system is needed to provide isolation and least privilege at the intra-process level to meet this new threat model. Crucially, this system must include defenses against native code to provide complete protection.

3.3 Threat Model and Security Goals

We seek to provide an intra-process privilege separation system, which allows developers to prevent third-party code included in their IoT applications from leaking data. In particular, Pyronia aims to provide the comprehensive yet efficient protection of sensitive OS resources, as well as restrict access to remote network destinations.

3.3.1 Threat Model

We assume that IoT device providers, who are usually also the application developers, are trusted. As such, we also trust the underlying device hardware, the operating system, and any language runtime executing the IoT application. Yet, imported third-party code poses a risk to developers: library code is rarely inspected or readily accessible (e.g., obfuscated), so bugs or dynamic language features that leak sensitive data may go unnoticed. We do, however, assume that application developers do not *intentionally* include such vulnerable or malicious third-party code into their products.

These threats are not IoT-specific. However, other compute settings (i.e., mobile, desktop, cloud) face more complex security challenges. Mobile devices and desktops (and also IoT hubs) run multiple mutually distrusting applications that need to be isolated. Further, desktops and cloud servers run in multi-user/multi-tenant settings that require additional separation between the different principals. In contrast, the majority of IoT devices are single-purpose running only a single application, with the primary threat stemming from untrusted third-party libraries. Pyronia's threat model thus applies most directly to IoT, so we focus on applying our approach in this setting.

While data leak vulnerabilities take many forms, Pyronia targets third-party code that aims to access arbitrary sensitive files or devices, or exfiltrate sensitive data to arbitrary remote network destinations. Pyronia does not seek to prevent any control flow (e.g., ROP [107]) or side channel attacks (e.g., Spectre/Meltdown [74, 81], or physical vectors [47]). Control flow attacks such as ROP attacks pose a challenge to Pyronia's memory isolation. Like previous proposals, ROP defenses (e.g., [39, 134, 60, 38]) may be used in a complementary fashion. Pyronia also does not prevent network-based attacks such as man-in-the-middle or denial-of-service attacks.

3.3.2 Security Properties

Pyronia provides four main security properties.

P1: Least privilege. A third-party library function may only access those OS resources (i.e., files, devices, network) that are necessary to provide the *expected* functionality. Attempts by a third-party function to access resources that are not relevant to its functionality must be blocked. Pyronia achieves this goal by conservatively enforcing a default-deny policy on a library function's access to OS resources, and requiring developers to explicitly grant specific library functions access.

P2: Data isolation. Security-critical language runtime metadata, most notably the call stack, may only be accessed by internal language runtime functions that operate on this metadata. That is, a third-party function cannot directly modify any security-critical metadata. Pyronia enforces this property by placing runtime metadata in strongly isolated memory domains. Access to these memory domains is only granted upon the creation of a new stack frame or local variable modifications, and is revoked after these functions return.

P3: Verified network destinations. A third-party library function may only transmit data to developer-approved remote network destinations, e.g., cloud servers or other IoT devices. Thus, a third-party library cannot leak legitimately collected data to an untrusted remote server or device. Pyronia prevents such data exfiltration by intercepting all outgoing network connections. Then, Pyronia verifies that all involved functions have sufficient privileges to transmit data to the requested network destination.

P4: No confused deputies. All resource access control decisions are made based on the *full provenance* of the access request. This prevents confused deputy attacks [55], in which an underprivileged library function attempts to gain unauthorized access to a sensitive resource via another function that does have sufficient privileges. To detect such attempts to bypass access control checks, Pyronia leverages the language runtime call stack to check all functions involved in a request for a privileged OS resource.

Non-goals. While Pyronia automates access control and data isolation at the level of in-application components, our design does not seek to provide automated execution isolation of these components (e.g., [129, 27, 26]). We also do not guarantee the *correctness* of the sensitive data they output. Automated compartmentalization is complementary to our approach, and could be added to Pyronia to allow developers to prevent certain cross-library function calls. Ensuring the correctness of sensitive outputs, on the other hand, could provide additional data leak protection for cases in which a vulnerable or malicious library designed to process sensitive data simply returns the unmodified input, or embeds other sensitive information as part of its output. However, verifying the functionality of untrusted code is beyond the scope of Pyronia, and could be performed as an additional step prior to application deployment.

3.4 System Design

Pyronia enforces intra-process least privilege without partitioning an application into multiple processes (e.g., [26, 27, 115, 139]), or propagating data flow labels (e.g., [44, 48, 140]).

Developers understand the purpose of imported libraries and can provide high-level descriptions of their expected data access behaviors. Pyronia thus relies on developers to

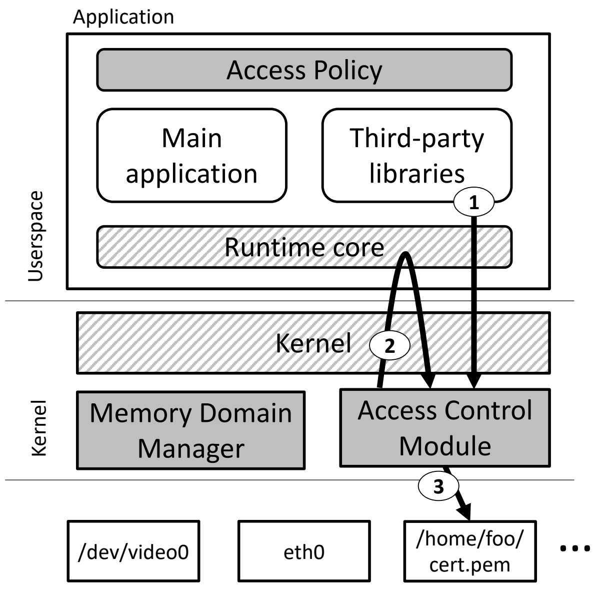

Figure 3.1: Overview of Pyronia, which enables the enforcement of a developer-supplied access control policy via runtime core and kernel modifications (striped boxes). New features are represented by gray boxes. The arrows show the components involved in an access request to a file such as a certificate.

specify all access rules in a single, central policy file. At run time, Pyronia loads this file into an application-specific access control list (ACL) that contains an entry for each developer-specified library function and its associated data access privileges. Pyronia imposes *default-deny* access control semantics, meaning that a third-party library function may only access those files, devices, and remote network destinations to which the developer has granted explicit access.

To enforce this policy securely, we modify the underlying language runtime core to load developer-supplied access control policies, and add an Access Control module and a Memory Domain manager inside the kernel. Figure 3.1 provides an overview of the Pyronia system architecture, and shows the main steps involved in a resource access request (see §3.4.1).

3.4.1 Function-granular MAC

At first glance, performing library function-granular access control for files and devices in the language runtime may seem like a sufficient solution. The runtime can directly inspect the call stack when a third-party library function uses the language's high-level interface to access a sensor or file. If the runtime identifies a function not permitted to access the resource, it can block the request and throw an error to notify the system.

However, language runtimes also provide an interface to native code, such as Java's JNI or Python's C API; indeed, our analysis in §3.2.2 shows that use of this interface in Python is very common-place. This ability to include native code in otherwise memory-safe languages exposes applications to vulnerabilities in native code: as this code runs outside the purview of the language runtime, this code could bypass any runtime-level OS resource access interposition via direct system calls.

Algorithm 1 Call stack inspection

```
1: procedure InspectCallstack($callStack$, $requestedAccess$)
2:     $accessRule \leftarrow$ GetAclRule($callStack.funcName$)
3:     if $accessRule ==$ nil then return false
4:     $grantAccess \leftarrow$ HasPrivileges($accessRule, requestedAccess$)
5:     while $grantAccess ==$ true do
6:         $callStack \leftarrow callStack.Next$
7:         $accessRule \leftarrow$ GetAclRule($callStack.funcName$)
8:         if $accessRule ==$ nil then
9:             continue
10:        $grantAccess \leftarrow$ HasPrivileges($accessRule, requestedAccess$)
       return $grantAccess$
```

Pyronia addresses these issues via an improved system call interposition technique that incorporates function call provenance. Many mandatory access control (MAC) systems in deployment use system call interposition (e.g., SELinux [112], AppArmor [14], Windows MIC [91]). Their limitation, however, is that the security policy is only enforced at the process level. We extend this mechanism to the intra-process level by adding runtime call stack inspection (e.g., as in [113, 41, 131]) when a system call is intercepted so that Pyronia can check the full provenance of the system call.

Thus, Pyronia employs these two techniques to provide *function-granular* MAC and enforce least privilege (**P1**). As we show in Fig. 3.1, when the application attempts to access a sensitive OS resource (e.g., an SSL certificate), the Pyronia Access Control module in the kernel intercepts the associated system call (step 1). This kernel module then sends a request to the language runtime via the trusted *stack inspector* thread, which pauses the runtime. After collecting the interpreter's function call stack in its current state, the stack inspector sends the stack back to the kernel (step 2).

The Access Control module maintains the access control list (ACL) for the developer-supplied policy. To determine whether access to the requested OS resource should be granted, the kernel inspects the call stack to obtain the full provenance of the system call. Only if all developer-specified functions identified in the call stack have sufficient privileges may the application obtain data from the requested resource (step 3). That is, to determine whether to grant a function call access to a requested resource, Pyronia dynamically computes the intersection of the privileges of each function in the call stack using algorithm 1. This algorithm prevents confused deputies (**P4**), much like in [113, 41, 43].

3.4.2 Runtime Call Stack Protection

While Pyronia's MAC system effectively enforces fine-grained least privilege for OS resources, untrusted native libraries still reside in the runtime's address space, giving them unfettered access to the call stack's memory region. A malicious native library may tamper with the runtime call stack in an attempt to bypass Pyronia's security checks.

This challenge is not unique to Pyronia; indeed, prior work in the mobile space [113, 133] recognized the need to protect the Dalvik call stack against native third-party code in the trusted host app's address space. To address this issue, these proposals either rely on special hardware support [113] to separate the runtime address space from the native library address space, or they forgo memory protection altogether [133].

Pyronia, in contrast, aims to provide a more generally applicable single-address space memory isolation mechanism, since IoT software runs on a very diverse range of hardware platforms. We overcome this challenge with page table replication, a technique that enables us to create strongly isolated memory regions within a process' address space. Prior work in this space (e.g., [132, 59, 82]) has introduced new primitives for intra-process execution compartments with strong isolation akin to that of process isolation. Our design for Pyronia's memory domains, on the other hand, focuses on *data isolation*.

Memory domains in Pyronia must meet two requirements: (1) the size of a domain must be flexible, and (2) the access privileges must be dynamic. The first requirement is important for ensuring that Pyronia can support arbitrarily complex applications that make many nested function calls. The second requirement allows Pyronia to restrict an application's access to a memory domain at run time based on the currently executing code (e.g., interpreter versus third-party library), while still enabling data sharing between application components that may need access to the protected data at different times.

The Pyronia Memory Domain manager in the kernel (see Fig. 3.1) maintains a per-process table of domain-protected memory pages, and replicates the corresponding page table entries specifying the associated accessibility bits for each memory domain access policy. Upon an access to a memory address, the Pyronia kernel performs all regular memory access checks; if the requested address is domain-protected, the Domain manager additionally verifies that the process has sufficient privileges to access the requested memory domain based on the loaded page entry. Any attempt by an application to access unauthorized domain-protected data results in a kernel memory fault, enforcing strong data isolation.

To ensure the integrity of the runtime call stack, Pyronia transparently allocates all call stack data into a memory domain, which we refer to as the *interpreter domain*. Stack frame creation and deletion functions are then run in an intra-process *dynamic execution sandbox*: the Pyronia runtime makes an access privilege adjustment call to the Domain manager, which loads the page table with the appropriate entry enabling the runtime to allocate or modify interpreter domain data (**P2**). Beyond these events, the language runtime still provides *read* access to its call stack to allow untrusted third-party code to make use of shared functions, but prohibits *write* access to protect this security-critical metadata.

3.4.3 Network Destination Verification

To prevent data leaks, Pyronia must additionally ensure that only authorized third-party library functions may transmit data to whitelisted network destinations. That is, when an application attempts to export data to a remote network destination, Pyronia's Access Control module intercepts all outward-facing socket system calls (e.g., *bind()* and *connect()*) for all socket types, i.e., TCP, UDP, and raw. As with other OS resource accesses, Pyronia then requests and inspects the runtime call stack.

However, network access privileges alone do not immediately allow a third-party function to transmit data. Pyronia also verifies the remote endpoint for the requested socket. Thus,

Table 3.5: Pyronia data access rule specification formats. Supported access privileges are read-only r, and read-write w.

Rule type	Format
OS resource	`<module>.<function name> <path to resource> <access privs>`
Network destination	`<module>.<function name> network <IP addr/prefix>`

only if the address of the requested remote network destination is included in the access rules for the given third-party function, does Pyronia grant access to the requested socket (**P3**).

3.5 Implementation

To demonstrate the practicality of Pyronia, and how it interacts with existing open-source IoT applications, we implemented the Pyronia kernel based on Linux Kernel version 4.8.0+, and the Pyronia runtime as a modified version of Python 2.7.14. We have released all components of our prototype on Github.[4]

3.5.1 Policy Specification

Developers in Pyronia specify all library function-level data access rules for their application in a single policy file. These rules are specified for files/devices and network destinations. Table 3.5 details Pyronia's policy rule specification format. In the case of network access rules specifically, our Pyronia prototype allows developers to specify IP address prefixes, as the specific address of a remote destination in a cloud service may not be known a priori.

Nevertheless, a challenge that arises from having limited knowledge about a library's implementation is that it may legitimately require access to other resources of which the developer is unaware or that may be unexpected. For example, a library function may need access to system fonts, or may write an intermediate result to the file system. Similarly, developers are unlikely to have a good sense of the system libraries or other file system locations a language runtime requires to operate properly. Thus, Pyronia's default-deny access control semantics alone are too restrictive and may lead to a number of false negatives. To maintain the functionality of the application, Pyronia needs to allow the runtime to perform necessary background operations.

Our solution to reducing the number of false negatives is to identify sane defaults that preserve the functionality of the Pyronia runtime and most applications, in the face of such execution "side-effects". In particular, based on three open-source IoT applications we evaluated (see §3.6.1), we identified a set of default system files and libraries (e.g., the `\etc\hosts` file, or `libdl.so`). These system files are required for the Python runtime itself, and hence may not be accessed as part of Python code execution. In other words, a runtime call stack may not exist at the time of access.

To allow the interpreter to access these system files at any point, our prototype supports a special *default* access rule declaration, which grants application-wide access to a specified

[4]https://github.com/princetonsns/pyronia

Algorithm 2 Pyronia in-kernel access control check

```
1: procedure CheckResourceAccess($acl$, $requestedAccess$)
2:     $isDefault \leftarrow$ IsDefaultAccess($acl.resource$)
3:     if $isDefault ==$ true then return true
4:     $loggedStackHashes \leftarrow$ GetLoggedStacks($acl$)
5:     if $loggedStackHashes != \emptyset$&&ReceivedStackHash($acl.resource$) == true then
6:         $loggedHashFound \leftarrow$ VerifyReceivedStackHash($loggedStackHashes$)
7:         if $loggedHashFound$ ==true then return true
8:     $callStack \leftarrow$ RequestCallStackFromRuntime return InspectCallstack($callStack, requestedAccess$)
```

resource. While default access rules bypass Pyronia's function-level access control, they still provide a baseline level of security as Pyronia also applies default-deny access control semantics at the application-level.

Our current Pyronia prototype requires that all default access rules be individually specified in the developer's policy. However, we facilitate policy specification for developers by providing a policy generation tool that creates a policy template file that already includes all identified Python runtime default system files. A future iteration of Pyronia could maintain a list of the most critical default rules within the runtime or even the kernel, allowing developers to remain agnostic to the runtime defaults.

3.5.2 Pyronia Kernel

Access Control module. Our Access Control Module extends the AppArmor [14] kernel module version 2.11.0. AppArmor interposes on all system calls enforcing a process-level access policy. Thus, as in vanilla AppArmor, Pyronia denies access to a requested resource if the *process* does not have sufficient privileges. To add support for Pyronia's stack inspection, we introduce a function-level access control list (ACL) to the process-level AppArmor policy data structure. This ACL is populated at application initialization (see §3.5.3), and contains an entry for each developer-specified library function. In addition, the Access Control module registers child processes of the main Pyronia process, propagating the application's function-level ACL to all child processes.

In an early version of our prototype, Pyronia would inspect the runtime call stack on *every* intercepted system call. However, given that IoT applications are built to continuously gather and transmit data in an infinite loop, we found that we could improve the performance of Pyronia's function-level MAC avoiding expensive kernel-userspace context switches for previously-verified call stacks.

To address this challenge, we implement a *stack logging* mechanism as part of the function-level MAC checks in the kernel, outlined in algorithm 2. Once the Access control module has verified the process-level permissions, the module checks the defaults ACL for the application (line 2). If the requested resource is covered by a default rule, Pyronia grants access without inspecting the call stack.

Otherwise, the Access Control module next checks the call stack log in the ACL for the requested resource (line 4). To facilitate verification of logged call stacks, our Pyronia prototype stores the SHA256 hash for up to 16 distinct function calls authorized to access a given resource. In the event that the ACL contains logged call stacks, the Access Control module checks if the Python runtime has included a call stack hash along with the system call (more details in §3.5.3). If the runtime has sent the kernel a call stack hash, the next step is to verify that the received hash matches any of the logged hashes (line 6).

If the Access Control module cannot find a matching logged hash, or if the ACL for the requested resource does not contain any logged call stacks, our prototype resorts to the full stack inspection mechanism (line 9, see Alg. 1). Once the call stack has been inspected, and if the application has sufficient privileges to access the requested resource, the Access Control module logs the SHA256 hash of the callstack in the resource's ACL.

Memory Domain management. For Pyronia's memory domains, we modify the SMV [59] memory module, an intra-process memory isolation proposal which adds support for enforcing per-page thread-level access control policies via thread-local page tables. While the main goal of SMV is to provide isolation for multi-threaded applications, this system still offers the appropriate abstractions to implement Pyronia memory domains. Our Memory Domain manager leverages the SMV kernel API for maintaining a list of protected 4kB domain pages and their corresponding memory domain access policy for each running thread (in Pyronia, the main and stack inspector threads, each with their own domain access policy).

To enable communication between the kernel and the runtime in userspace, Pyronia uses two generic Netlink sockets to allow communication with the Domain manager, and the Access Control module, respectively. Netlink sockets offer two advantages: (1) they allow bi-directional communication between kernelspace and userspace obviating the need to implement additional ioctls() or system calls, and (2) userspace applications can use the same API for Netlink communication as for regular socket-based communication.

3.5.3 Pyronia Python Runtime

To allow developers to run completely unmodified applications, the Pyronia runtime acts as a drop-in replacement for Python. We integrate our Pyronia userspace library, which provides an API for automatically loading the developer's access policy, as well as a memory domain management API.

Policy initialization. The runtime core uses our policy parser API to read the developer's policy file during interpreter initialization. All parsed OS resource and network access rules are sent to the Access Control module initializing the function-level ACL for the application.

Loading the policy at this step before the runtime has loaded any third-party code also has the advantage of preventing an adversary from "front-running" the interpreter by initializing the application's in-kernel Pyronia ACL before the legitimate developer-supplied policy can be loaded. For this reason, the Pyronia runtime also spawns the stack inspector thread and registers it with the kernel during the initialization phase.

Memory domain allocation. The userspace Pyronia memory domain management API acts as a drop-in replacement for malloc. To allocate the runtime call stack in the *interpreter* *domain*, we instrumented the Python stack frame allocator. Because write access to the interpreter domain is disabled by default, the runtime temporarily obtains write access to this domain during frame creation and deletion operations.

Forking child processes. Our Pyronia prototype provides continuous protection for child processes spawned via standard Python APIs (e.g., *os.system()*). To provide this continuity, the Pyronia initialization function keeps track of the main application process' PID, in order to detect when the runtime forks a child process.

As forking preserves the parent Pyronia process memory in the child, Pyronia child processes automatically inherit the parent's memory domain layout as well as the Pyronia interpreter metadata, including currently writable memory domains. Additionally, Pyronia child processes inherit the parent's function-level ACL in the Access Control module. Thus, Pyronia initialization in child processes only needs to spawn the child's stack inspector thread, and reset the application's access permissions to the interpreter domain disabling write access to this domain. This reset is necessary to ensure that the child begins its execution in a clean state, and cannot access any interpreter stack frames in its own address space that the parent process had marked as writable at the time of forking.

Importantly, initialization of Pyronia occurs automatically in child processes forked using standard Python APIs such as *os.system()*, allowing the continued protection of child processes without developer intervention.

Stack logging. As described in §3.5.2, we implement a stack logging mechanism to reduce the overhead of kernel upcalls for runtime call stack requests. When the language runtime requests a resource for the first time, the system call proceeds as usual. If the kernel allows the system call to complete, the Pyronia runtime then logs the authorized OS resource. Then, in subsequent system calls, our prototype first checks if the requested OS resource has been previously logged as authorized. If this is the case, instead of waiting for an upcall from the kernel, the runtime preemptively collects its current call stack *before* making the system call. The runtime then computes the SHA256 hash of the call stack, and serializes the result as part of the input into the upcoming system call.

To enable this optimization, we created system call-specific wrappers for various incarnations of `open()` and `connect()`. These wrappers perform the preemptive call stack collection, hash serialization, and authorized OS resource logging. On the kernel side, we made minor modifications to the `open()` and `connect()` kernel counterparts to parse the received call stack hash, if any, and store it for later verification during the MAC checks.

Both our userspace stack logging wrapper functions, as well as the instrumented system call kernel code, are backwards compatible to handle conventional system calls. We ensure that all targeted system calls are wrapped via LD_PRELOAD. Notably, stack logging did not require us to modify the data access policies for our tested applications.

Garbage collection. One key feature of Python that poses a challenge is the object reference counting used for Python's garbage collection. Specifically, we found that Python increments or decrements several objects' reference counts, including those of call stack frames, for practically every Python instruction and internal operation.

To maintain the functionality of the Pyronia runtime without unduly weakening data isolation, we added interpreter domain write access adjustment calls around those blocks of the Python runtime code that operate on domain-protected data. Yet, simply granting write access to an entire memory domain is inefficient, especially for the interpreter domain, which may contain hundreds of protected pages, each of whose page table entries would need to be modified (requiring TLB flushes at high rates).

Pyronia's memory domain implementation provides us with an opportunity to optimize these frequent domain access adjustments. Because memory domains enforce per-page access control, we can limit each adjustment to a single page given the address of the stack frame or other domain-protected metadata that the interpreter needs to access.

To this enable this optimization, the Pyronia runtime thus keeps a list of pages in the interpreter domain, and only modifies the access privileges to a specific page when needed. One exception to this is for new stack frame and local variable dictionary allocations. In these cases, since the runtime is creating a new buffer with an undetermined memory address, our prototype enables write access to all domain pages with free memory chunks.

3.6 Pyronia Protection in Real Applications

To examine the effectiveness of Pyronia's policies and protections in real applications, we conduct three in-depth case studies of Python applications that specifically capture a range of common IoT use cases and import a variety of Python libraries. Our goal is to answer the following questions about the usability and security of Pyronia:
§3.6.2 How difficult is it to write function-level access policies for Pyronia?
§3.6.3 What reported Python library vulnerability classes can Pyronia mitigate?
§3.6.4 Is Pyronia robust against attempts to circumvent function-level MAC?
§3.6.5 What effect do dynamic language features of Python have on Pyronia's protections?

3.6.1 Case Studies

We evaluate three open-source Python IoT applications that represent the main categories of applications we studied: visual, audio, and environmental sensing. Each of these applications communicates with a cloud service for data processing or storage, which required that we register an account to obtain authentication credentials. The libraries in the applications we choose implement a broad range of common IoT functionalities with different properties. Specifically, our goal is to study how Pyronia operates for common IoT authentication mechanisms, data processing techniques, and communication protocols.

Through manual inspection of the source code of each case study, we found four distinct direct *third-party* imports, and three standard libraries. While small in number, these direct imports are all among the top 50 overall or top 50 third-party imports in our IoT application analysis (§3.2.2). Table 3.6 gives an overview of our case studies.

`twitterPhoto` takes an image from a connected camera every 15 minutes, and sends the picture along with a short message as a tweet to a specified Twitter account. Before sending the tweet, the application authenticates itself to Twitter via OAuth. The tweepy library is used both to authenticate the app and upload the Tweet. We obtained this application from the set of 85 applications described in §3.2.2.

Table 3.6: Summary of case study applications.

App	3p imports	Highlighted features	# func-level rules
`twitterPhoto`	tweepy	integrated OAuth, HTTP	5
`alexa`	json	data marshalling	0
	memcache	raw sockets	1
	re	regex parsing	0
	requests	widely used HTTP API	10
`plant_watering`	paho-mqtt	MQTT comm, binary exec	11
	ssl	crypto, native dependencies	0

`alexa` aims to provide an open-source Python implementation of an Amazon Echo smart speaker. This application records audio via a microphone while a button is pressed, and sends the recorded audio (along with the required authentication credentials) to the Alexa Voice Service (AVS) [12] for processing. The AVS sends an audio response, if the recorded data is one of the commands recognized by the service, for the `alexa` application to play back. Otherwise, the AVS responds with an empty message acknowledging the receipt of the recording, in which case the application does not play back any audio. This app uses the python-memcache library to cache the app's AVS access token, and the requests library to communicate with AVS via HTTP. This app also uses the json standard library to format all messages exchanged with AVS, and the re standard library to parse out any audio contained in the AVS response. To facilitate our security and performance tests, we have removed the need for the button press for audio recording, and instead open a static pre-recorded audio file. This application is also among the 85 applications we analyze in §3.2.2.

`plant_watering` records moisture sensor readings once a minute, and sends them to the Amazon AWS IoT [9] service via MQTT [3], a widely used IoT messaging protocol. MQTT also handles client authentication with the AWS IoT service via TLS. As above, we have replaced sensor readings with a randomly generated value. We adapted this application from an AWS IoT SDK tutorial [10]; because our Pyronia prototype does not currently support multi-threading, `plant_watering` uses the Python Paho MQTT library [4], which supports single-threaded network communications, in place of the original application's use of the AWS IoT Python SDK [1], which only supports a multi-threaded MQTT client.

3.6.2 Specifying Function-Level Policies

While Pyronia avoids application refactoring and code annotation, developers are still required to specify a fine-grained access policy for their application. To understand whether Pyronia's policy specification places an undue burden on developers, we analyzed the policy specification process for our three case studies.

As we describe in §3.5, one challenge to specifying comprehensive rules for a MAC system that enforces default-deny semantics is reducing the number of false negatives. In running our case studies, we found that the vast majority of false negatives would arise primarily due to Python's internal library loading and DNS resolution processes.

To address this challenge, we ran each case study application, in Pyronia under AppArmor in complain mode, and identified a total of 42 common files that need to be read-accessible, plus 4 network protocols, for the Python interpreter to run unimpeded, and for the applications to connect to remote network destinations. Our prototype's reliance on AppArmor requires us to specify a separate rule for each file and network protocol. We show the resulting AppArmor policy rules in Appendix C.

In addition, each application required explicit access to its parent directory as well as all imported Python modules contained within, which corresponds to up to 6 additional access rules for the `alexa` application. Since we cannot expect developers to manually specify around 50 access rules needed by their application by default, we provide a policy generation tool that creates an access policy template pre-populated with rules for the 46 common files and network protocols. To further ease policy specification, our policy generation tool lists all files in the application's parent directory and adds rules for all identified files. Developers may then manually inspect the template and modify any rules that are function-specific.

Table 3.6 summarizes the number of application-specific access rules for each case study. The majority of the manually added access rules for all three applications are multiple network destination rules for the same library function. Two limitations of our Pyronia prototype account for this. First, Pyronia does not currently support domain name-based access rules as this feature requires that all domain names be resolved a priori for Pyronia's IP-address based network destination verification. As a result, developers must currently specify a separate network destination rule for each IP address prefix their application connects with. Nonetheless, our case studies require no more than 11 function-level network destination rules in the `plant_watering` case.

These numbers could be further reduced by adding support for rule grouping. That is, for resources that are accessible by multiple functions with the same access privileges, Pyronia could support allowing developers to express these rules within a single rule. In the case of the `alexa` app, for instance, which currently requires 10 network access rules because two functions in the requests library connect to the same 5 network destinations, the number of required rules could be reduced by half with rule grouping, and even more with support for domain name-based rules. Appendix C shows the function-level policy rules for each of our case studies.

In summary, false negatives are the primary concern for Pyronia's policy specification. We believe that default rules, coupled with our policy generation tool, significantly reduce the number of false negatives application developers encounter.

3.6.3 Vulnerability Analysis

To understand how applicable and effective Pyronia's protections are against more general security vulnerabilities and exploits, we study specific instances of reported Python library vulnerabilities (recall §3.2.1), and analyze Pyronia's ability to mitigate these cases. We emphasize that all of our analyzed vulnerabilities could affect *any* IoT application, so our choice for testing a particular vulnerability in a specific application does not reflect prevalence of the tested vulnerability class in a specific IoT use case.

Since most of the reported vulnerabilities do not affect the libraries in our case study applications, we replicate all analyzed vulnerabilities in a specially crafted adversarial library targeting our case studies. We then call individual functions in our library from our case study applications, and modify the corresponding access policies to allow our adversarial functions to access the necessary OS resources and network destinations.

We place the 9 reported attack classes into three broad categories: (1) mitigated for attacks that Pyronia successfully prevents, (2) case-dependent for vulnerabilities that Pyronia may mitigate in some instances, and (3) beyond scope for attacks that fall outside Pyronia's threat model.

Direct data leaks. We analyze three distinct instances of OS resource-based data leaks, in which code with insufficient privileges attempts to gain access to a sensitive OS resource. Specifically, we craft an adversarial function for the `twitterPhoto` app that wraps the tweepy library function call to upload image file, and attempts to upload an SSH private key instead of the authorized photo file using the legitimate tweepy library call. Similarly, we implement a function that attempts to upload the authorized photo to an unauthorized remote server. In addition, we also replicate the data leak bug reported in CVE-2019-9948 [2], in which the Python urllib HTTP library exposes an unsafe API that allows arbitrary local file opens.

The reason we classify direct data leaks mitigation as case-dependent in Pyronia is because several reported data leak vulnerabilities arise due to in-memory data exposures, as Pyronia only ensures the protection of the runtime call stack memory, but no other sensitive application-level in-memory data objects.

Symlink attacks. A small number of reported vulnerabilities in Python libraries comprises symlink attacks, in which an adversary attempts to access an unauthorized file via a specially crafted symbolic link. We analyze this attack by crafting an adversarial function that attempts to open `plant_watering`'s private key through a symlink in the `\tmp` directory. Since our prototype follows all accessed symbolic links to their source, Pyronia detects the attempt to access an unauthorized file successfully mitigating this type of attack.

Arbitrary code execution. A number of reported Python library vulnerabilities pertain to shell injection made possible due to unsanitized input. To examine this attack class, we crafted an adversarial library function that attempts to directly exec a shell command as part of the `plant_watering` moisture sensor reading (i.e., random number generator) call.

As in the direct data leaks case, Pyronia successfully mitigates the exec. While none of our case studies are susceptible to unsanitized input bugs, we believe that our analysis demonstrates that Pyronia could be effective in mitigating all instances of shell injection attacks as they all ultimately require access to an unauthorized executable binary file.

Also similar to the direct data leaks case, Pyronia does not mitigate memory-based arbitrary code execution attacks. Such attacks typically leverage a buffer overflow vulnerability that allows an attacker to hijack the control flow of the victim application. Further research is necessary to determine if Pyronia's memory domains could mitigate these attacks.

Beyond scope vulnerability classes. A large portion of reported Python library vulnerabilities are beyond the scope of Pyronia's protections. The majority of MITM vulnerabilities in our CVE reports analysis stem from a failure to verify TLS certificates. The weak crypto vulnerabilities in our survey primarily arise because a known-weak algorithm or random number generator was used, or because input was not properly validated. Similarly, Pyronia cannot prevent replay or data spoofing attacks as these stem from improper input (i.e., nonce or filename) validation as well. Most of the authentication bypass bugs occur in larger Python-based frameworks that fail to properly implement authentication procedures.

Interestingly, the majority of the reported DoS attacks in Python libraries stem from improper input handling or memory management that can cause the application to crash. While Pyronia does not verify the correctness of application or library code, we see an avenue for using Pyronia to prevent network-based DoS attacks.

3.6.4 Limiting MAC Check Circumvention

Our analysis in §3.6.3 demonstrates that Pyronia can successfully mitigate several classes of OS resource-based vulnerabilities. However, we also wanted to evaluate how robust Pyronia is to attempts by more clever adversaries to circumvent our function-level MAC mechanism. As we describe in §3.4.1, a library function without privileges to access a given resource may attempt to fool the MAC checks by calling a more privileged library function that does have sufficient privileges to access the desired resource.

Thus, we analyzed two specific instances: (1) an entirely unprivileged library function (i.e., it does not appear in any rules in the application's access policy) calls an authorized function, and (2) a function authorized to access an OS resource calls a function with privileges to a different resource. Because Pyronia's stack inspection mechanism in the kernel always begins by inspecting the outermost caller (i.e., the bottom of the stack), it will immediately return to the a "permission denied" error to the application, as the stack inspector cannot find an ACL entry for the outermost function call in the stack.

3.6.5 Pyronia Under Dynamic Language Features

Prior proposals have recognized the potential security threat posed by dynamic language features such as reflection and native code execution [113, 129]. As we describe in §3.2.1, Python's dynamic features enabled us to replace function pointers (aka monkey patching), leak a sensitive file at import time, and modify contents of Python stack frames including the value of function arguments. To understand how these dynamic language features affect Pyronia's protections, we analyzed these three cases, as well as external binary execution.

Much as in the direct data leak attack analysis, we found that Pyronia readily prevented unauthorized file accesses at import time. We also found that external binaries executed in Pyronia child processes were unable to access files and network resources not authorized by the parent's data access policy.

However, we met several challenges when analyzing Pyronia's ability to prevent all forms of stack frame tampering and monkey patching. We found that Pyronia's memory domains prevent native code from directly accessing arbitrary stack frame memory. Yet, because Python stores the local variables for each stack frame in a separate dictionary data structure, pointed to by the stack frame, we found that our implemented stack frame isolation is insufficient to prevent tampering with function arguments by native code or monkey patching. As part of ongoing research, we are exploring more robust countermeasures to these dynamic language features.

Table 3.7: Mean end-to-end execution time overhead for the first app iteration, and for subsequent iterations (long-term), for each case study application. With the stack logging optimization in place, this measurement represents the worst-case execution time.

	e2e exec time	**per-iter overhead (e2e)**	**max stack depth**
`twitterPhoto`	1.5s	3x (5x)	14
`alexa`	1.2s	2x (5x)	14
`plant_watering`	0.2s	3x (2x)	5

3.7 Evaluation

To evaluate the performance of Pyronia, we ran our three case studies (§3.6.1) in vanilla Python and Pyronia Python, measuring the execution time and memory overheads. We also took microbenchmarks of the main Pyronia operations, as well as common system calls used in IoT applications with and without Pyronia enabled to analyze the impact of Pyronia's call stack inspection-based access control.

Our testing system is an Ubuntu 18.04 LTS virtual machine running our instrumented Pyronia Linux Kernel on a single Intel Core i7-3770 CPU clocked at 3.40 GHz, with 1.95 GB of RAM. Though not a dedicated IoT platform, our test VM's configuration is comparable to recent single-board computers targeting IoT, such as the Raspberry Pi 4 [106] or the NVIDIA Jetson Nano [96].

To ensure that the results of our evaluation are consistent, we make minor modifications to our case study applications replacing their real-time data collection (e.g., reading an image from a camera) with a static data source (e.g., an image file), and run the applications for a finite number of iterations, replacing the infinite loop that keeps the applications running continuously when deployed on an IoT device. We emphasize that none of these modifications were necessary to specify a library function-level data access policy, or to add support for Pyronia's security mechanisms.

3.7.1 Execution Time Overhead

To analyze the impact of Pyronia on application execution time, we measured the end-to-end application execution time for a single iteration, as well as the per-iteration execution time over 100 iterations. The end-to-end execution time measurement for a single iteration of the application represents the worst-case execution time as it includes any overhead due to Pyronia initialization (and teardown), and the kernel's call stack log is empty. Measuring the per-iteration execution time, on the other hand, gives an estimate of normal operating time of the application, and the overhead due to Pyronia's run-time security checks, i.e., call stack inspection and interpreter domain-related operations.

Table 3.7 shows the mean end-to-end execution time overhead over 25 runs of a single iteration of our tested applications. While Pyronia's end-to-end execution time overhead of 2-5x is significant, the mean long-term overhead per iteration is reduced to 2-3x. Nonetheless, in absolute terms, the longest end-to-end execution time for a single app iteration under Pyronia for our case studies is 1.5 seconds, which we expect would remain largely imperceptible to end users in real-world deployments.

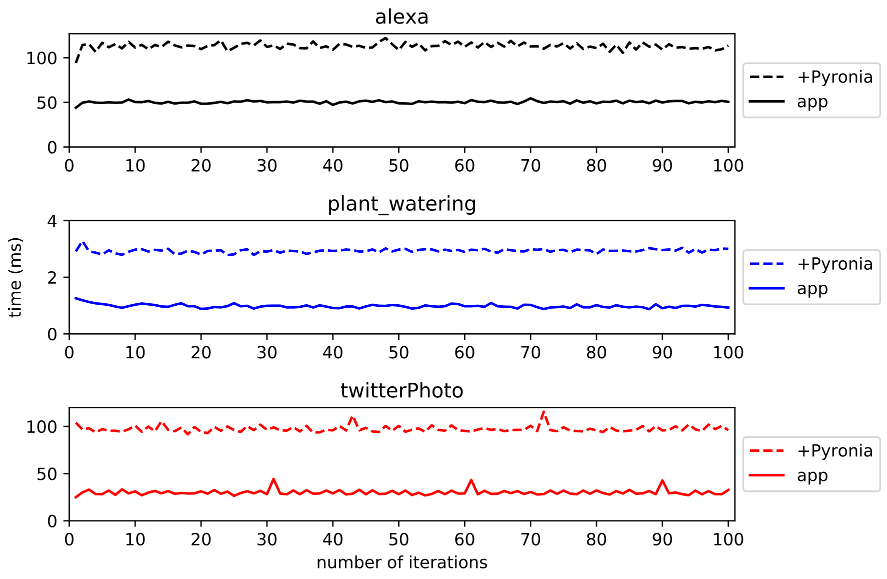

Figure 3.2: Mean per-iteration execution time in seconds for each application with and without Pyronia enabled.

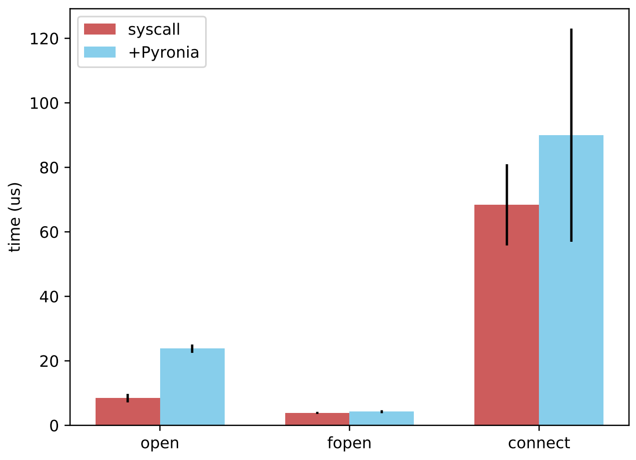

Figure 3.3: Mean execution time in microseconds for open, fopen and socket connect system calls across all tested applications.

Figure 3.2 plots the mean per-iteration execution time over 5 runs of 100 iterations our tested applications. Despite the overall execution time overhead of Pyronia, we observe that stack logging does lower the long-term overhead of the `plant_watering` and `twitterPhoto` apps, as the execution time mostly levels after about 5 iterations.

Nonetheless, stack logging seems to play a little role in reducing the execution time overhead for the `alexa` app, as the first-iteration and long-term per-iteration overhead is 2x in both cases. For an application that requires active user involvement, Pyronia's long-term per-iteration execution time overhead of 2x for the `alexa` app is likely to be unacceptable in a real-world deployment, even with an absolute per-iteration execution time of under one-eighth of a second (113.3 ms).

Pyronia operation microbenchmarks. To gain a better understanding of the source of Pyronia's overheads we ran microbenchmarks of the main 9 Pyronia operations over 25 runs of a single iteration of our case studies. Our results show that interpreter domain dynamic access adjustments greatly dominate the Pyronia overhead, with the median number of access grant calls of over 1 million for all tested applications. The significantly higher execution time for the `alexa` app can perhaps be attributed to the over 9 million interpreter memory domain access grant calls recorded in our experiments. Performance improvements for interactive IoT devices remain as future work.

Access control overhead. To further characterize the performance costs due to Pyronia's access control checks, we ran microbenchmarks of the libc *open()*, *fopen()* and *connect()*. We choose to measure these system calls as we have implemented our stack logging optimization for these system calls (and their 64-bit counterparts).

Figure 3.3 shows the mean execution time over 25 runs of a single iteration of our tested apps. Our results are consistent with our Pyronia operation microbenchmarks discussed above, showing that Pyronia's system call interposition imposes at most a 2x overhead for the *open()* system call.

Summary. Pyronia's execution time overhead is not trivial, despite our performance optimizations (see §3.5.3). While some additional time is spent during the each system call, the main slowdown occurs due to dynamic memory domain page access adjustments. Nonetheless, because the majority of IoT applications run on devices only passively collecting and transmitting data, we expect these execution time overheads would go largely unnoticed by end-users. We plan to investigate further performance optimizations of memory domain access adjustments, especially for interactive IoT applications, as part of future work.

3.7.2 Memory Overhead

In addition to execution time overhead, Pyronia imposes memory overhead as the language runtime maintains memory domains for the security-critical interpreter state (i.e., call stack and stack inspector), and for data object isolation. To evaluate the impact memory domains alone have on memory consumption, we first measure how much additional memory Pyronia uses to maintain each domain.

For a more accurate estimate of Pyronia's memory overhead due to memory domains, we focus our measurements on the userspace *per-domain page* metadata allocations, i.e., the memory required for the Pyronia runtime to maintain each domain page and the associated memory management data structures for individual allocated blocks within a page.[5] Table 3.8 shows the mean per-domain page memory consumption in bytes, as well as the median memory usage of the whole Pyronia subs-system in the Python runtime, over 5 runs of 100 iterations of each tested application.

We find that the mean Pyronia per-domain page memory consumption for all tested applications is between 0.31 and 0.38 KB; the fact that the page metadata allocations varies this little across all tested applications demonstrates that the majority of domain pages

[5] We do not evaluate the actual *data* allocation overhead per domain page as Pyronia does not change the amount of data the runtime allocates, only where in the runtime's address space this data is placed.

Table 3.8: Mean Pyronia per-domain page metadata memory usage for each application.

	# dom pages	per-dom page metadata	Pyronia total
`hello`	17	376 B	22.3 KB
`twitterPhoto`	176	309 B	151.2 KB
`alexa`	141	323 B	147.9 KB
`plant_watering`	54	361 B	68.4 KB

Table 3.9: Peak memory overhead under Pyronia for each application.

	peak usage (in MB)	overhead
`twitterPhoto`	52.7	2.2%
`alexa`	23.5	11.0%
`plant_watering`	18.8	28.1%

contain a similar number of allocated blocks (i.e., runtime stack frames), regardless of the total number of allocated domain pages. This result is a strong indication that Pyronia's domain memory consumption scales linearly with the number of allocated domain pages.

Our analysis also shows that the Pyronia sub-system in the Python runtime requires no more than a few hundred kilobytes of additional RAM for all tested applications (see Table 3.8). Furthermore, Pyronia's memory domains have a small impact on the overall memory consumption of our tested applications. Table 3.9 shows the mean peak memory usage and overhead in MB over 25 single iteration end-to-end runs. For the `twitterPhoto` application with a maximum memory usage of over 50 MB, Pyronia's memory overhead is only 2.2%, even with the largest number of data object domains.

Summary. Pyronia incurs low memory overhead, even for IoT applications that allocate hundreds of domain pages. For instance, domain metadata only consumes a total of 53 KB for the `twitterPhoto` application with 176 allocated domain pages. For applications with a greater number of domain pages, our results indicate that the metadata memory overhead would grow linearly. While the increase to application-wide memory usage is the highest for the `plant_watering` application at 28.1%, a peak memory consumption of under 20 MB is still rather modest. Therefore, Pyronia's memory overhead would not place an excessive burden on IoT devices with more constrained resource requirements than our testing system.

3.8 Discussion

Multi-threading. While our current Pyronia design targets single-threaded IoT applications, as we discovered in §3.2.2 as well as during our experiments, that IoT applications and libraries commonly spawn multiple pthreads. This application design pattern introduces one key security challenge: it breaks the continuity of the runtime call stack because threads execute independently of the main thread. In other words, a bug or vulnerability could still cause a confused deputy attack (breaking security property **P4**).

To address this issue, whenever Pyronia detects such continuity breaks (e.g., by interposing on `pthread_create()` or `thread_start()` calls), Pyronia could automatically inspect the state of the runtime call stack at that time, and save this "parent" call stack to provide the Access Control module with the full provenance when the child makes a system call. Nonetheless, ensuring that the Access Control module can properly map "parent" stacks to child stacks, especially in the scenario of nested multi-threading, would be an additional design challenge.

Preventing Pyronia API misuse. Pyronia currently gives the language runtime full authority over security-sensitive operations (e.g., memory domain allocations and privilege adjustments, or call stack generation), and trusts that only the runtime is making such Pyronia API calls. However, a resourceful adversary could use the API to bypass Pyronia's protections, or otherwise do harm, for instance by freeing domain-protected memory.

To prevent such misuses of the Pyronia API, we envision two potential approaches. In the style of [41], Pyronia could use cryptographic mechanisms to authenticate the Netlink messages the runtime (including the stack inspector thread) sends to the kernel. Another approach would be use static analysis to identify rogue Pyronia API calls in third-party code prior to launching the runtime. Further investigation is required to determine which method is more suitable.

Improving policy specification. As we discuss in §3.6.2, our very basic policy specification format and policy template generation tool aim to reduce the burden of defining fine-grained access policies with sane defaults. Nevertheless, Pyronia currently expects path-based access rules for OS resources in particular due to our reliance on AppArmor (see Appendix C). Having to determine the paths to the file system interface of resources such as sensors, likely still places an undue burden on developers. Designing a more developer-friendly and rigorous policy specification model is beyond the scope of Pyronia. One interesting approach may be to support simple mobile-style resource access capabilities (e.g., `READ_CAM`), which Pyronia could then automatically map to the corresponding low-level system resources.

While we believe that the risks of completely automated policy generation (e.g., as in [103, 27]) outweigh the benefits, we see an opportunity for the library developer community to ease the policy specification process further. For instance, library developers could contribute resource "manifests", i.e., a list of required files and network destinations, and package these manifests along with their source code or binaries. With support from Pyronia, application developers could then automatically load these manifests as part of their application-specific access policy, allowing application developers to focus on their high-level policy.

In-Memory data object isolation. While Pyronia's OS resource-based MAC mitigates several classes of data leak vulnerabilities, memory-based data leak attacks remain a serious concern. We see an opportunity to leverage Pyronia's memory domain mechanism used to protect runtime stack frames to extend Pyronia's protections to sensitive in-memory data objects as well.

We envision such data object protection sharing some aspects of language-level information flow control (IFC), while avoiding this approach's main limitations. Much like language-level IFC, developers would assign labels to sensitive function arguments and return values, but these labels would only be used as human-readable identifiers for sensitive in-memory data objects in the policy and by the Pyronia's runtime. For example, to identify the sensitive output object of the library function `recognize_face()`, the developer could use the label `face_image` in her policy, and use this label in any rules granting functions access to this same object. The Pyronia runtime could then automatically map the `face_image` label to the corresponding function argument or return value buffers. Further, these buffers would be assigned to a separate memory domain at run time, and the authorized functions run in an intra-process dynamic execution sandbox enforcing the developer's policy.

3.9 Related Work

Information Flow Control. Decentralized information flow control (IFC) systems explicitly label data and track how they flow through the system, such as an OS [42, 142, 76, 141], or a language runtime [37, 120, 48, 93, 140]. Principals (e.g., processes or functions) raise their labels when reading labeled data and invoke declassifiers when outputing sensitive information. Usually, restrictions are applied at disclosure time (e.g., filesystem write or network transmission) while no restrictions are imposed at read time. For example, Asbestos [42] allows any process to create new labels and raise the security clearance of other processes to facilitate reading of secret information. Pyronia's goal differs from that of IFC systems. Pyronia focuses on preventing function-level data access violations by transparently checking provenance information for syscalls, without requiring application developers to perform any unintuitive prior data flow analysis, or manual code annotations, required by IFC systems.

Memory Isolation. In addition to page table replication-based memory domains, a wide variety of alternative intra-process memory isolation techniques have been proposed. SFI [130], for instance, partitions a process' address space into fine-grained fault domains for different code blocks and data by modifying the program's binary. Mondrian memory protection (MMP) [135, 136] is a memory isolation system that enforces word-level access policies in the kernel.

Recent CPUs have introduced a number of hardware-assisted memory protection techniques that enable efficient in-application memory isolation. For example, shreds [36] leverages ARM memory domains [15] to create sub-thread-level isolated execution units. Intel SGX [68] allows developers to partition an application into a trusted and untrusted component, where the trusted component resides inside a cryptographically protected memory region within the application's address space. Despite their efficiency and ability to reduce the TCB size, these techniques require specialized hardware that is unavailable on most IoT devices. Pyronia, in contrast, aims to be more generally applicable.

Android-specific privilege separation. Several studies [28, 51, 121] showed how Android libraries, such as in-app advertising, collect private information by abusing permissions granted to their host app. Much work has been done to mitigate this privacy threat. Most prior approaches [101, 115, 122, 73, 58, 144, 61] separate third-party libraries by running them in isolated processes with different (usually lower) privileges from the host app. However, such process-based isolation suffers from high performance overheads imposed by additional IPC. In contrast, Pyronia does not rely on process isolation. We leverage stack inspection and memory domains to enforce least privilege and isolation within a process, avoiding any application refactoring. Furthermore, our approach of intra-process isolation fits better with existing hardware in the IoT ecosystem. Most IoT devices lack the hardware capability (e.g., a MMU) to support traditional process-based isolation. Instead, they often provide memory protection units (MPU) that allow simple access control configurations for regions of memory addresses. Pyronia's page replication-based memory domain mechanism can be implemented on devices with only MPUs where process-based isolation is fundamentally infeasible.

The FlexDroid [113] project is closest to our approach. FlexDroid also provides privilege separation by leveraging stack inspection and memory domains. However, FlexDroid's access control is built into Android's Permission Monitor service and its memory partitioning mechanism relies on special ARM architecture support.

IoT-specific Access Control. A number of recent works in the IoT space propose novel access control systems to protect sensitive user data against potentially vulnerable or malicious IoT applications. However, the majority of these systems focuses on enabling developers and end-users to define and enforce more suitable data access policies based on external factors such as usage context or risk [72, 111, 56, 105, 127]. Pyronia, in contrast, focuses on allowing developers to contain third-party code decoupling end-user application usage policy enforcement from a developer's implementation policy.

The FACT system [79] controls access to different IoT device functionalities by executing different IoT device functions in separate Linux containers. In contrast to Pyronia, however, FACT aims to prevent overprivileged *applications* from accessing sensitive device resources, but does not protect these resources against untrusted third-party code running as part of the isolated device functionalities, as Pyronia would. FlowFence [44] shares the same goals as Pyronia providing a framework for in-application privilege separation and data isolation. However, unlike our approach, FlowFence still relies on process isolation and does not support unmodified applications.

Python Isolation. Unfortunately, in the case of Python specifically, little has been done to restrict untrusted third-party modules.[6] A small number of projects has focused on securing Python against vulnerabilities and untrusted code via techniques such as information flow control [140], restricting Python's interfaces to privileged operations and introspection [32, 146], or process-based isolation [125]. However, unlike Pyronia, these works provide only coarse-grained protection remaining vulnerable to malicious libraries, do not control access to system resources, or rely on inconvenient process-level isolation.

3.10 Pyronia Summary

We have presented Pyronia, an in-application privilege separation system for language runtimes capable of integration into IoT applications. Pyronia protects sensitive data against

[6]According to the official Wiki on Sandboxed Python [104], proposals for modifying the CPython runtime to make applications more secure have been rejected because of their lack of usability or ineffectiveness against Python's introspection capabilities; instead a variety of ad-hoc mechanisms, some of which we discuss above, are recommended.

untrusted third-party libraries by leveraging three access control techniques —system call interposition, runtime call stack inspection, and memory domains. This design avoids the need for expensive process isolation or dynamic data flow tracking to enforce least privilege and provide strong data isolation at the granularity of individual library functions.

Further, unlike prior approaches, Pyronia runs unmodified applications, and does not require unintuitive policy specification. We implement a Pyronia kernel and Pyronia Python runtime. Our evaluation of three open-source Python IoT applications demonstrates that Pyronia mitigates OS resource-based data leak vulnerabilities, and shows that Pyronia's performance overheads are acceptable for the most common types of IoT applications.

Chapter 4

Griffin: Privilege Separation for Trusted Execution Environments

4.1 Introduction

Trusted execution environments (TEEs) such as SGX [68, 84] enable developers to create protected execution areas, called *enclaves* within a CPU. At its core, a TEE aims to provide confidentiality and integrity of sensitive application code and data in the presence of untrusted or vulnerable system software. In other words, the goal of TEEs is to allow developers to reduce the TCB of applications by placing only the most security-critical functionality in the enclave, and leaving the majority of application function the *untrusted* context.

Yet, deploying existing applications to TEEs typically requires that developers significantly re-architect their applications. In the case of SGX, for instance, applications may not make direct system calls from within an enclave. Thus, SGX application development has been shifting towards a containerized model, in which unmodified applications run on top of a TEE-specific library OS (libOS) inside an enclave [34, 118, 17, 21, 126]. These libOSes provide a general system call interface that transparently handles all enclave-to-untrusted transitions needed to call system software.

However, this practice vastly increases the trusted computing base (TCB) of an application, which now includes any untrusted third-party code as well as the underlying TEE libOS. Including third-party libraries within an enclave is problematic for two main reasons. First, application programmers rarely make security a priority in their development workflow [20, 6, 7]. Thus, developers cannot be expected to fully inspect the source code of every library they need for their application, meaning that any vulnerabilities or bugs in imported third-party software may remain undetected.

Nevertheless, the convenience of third-party libraries makes them indispensable to today's TEE software development practices. For example, in spite of numerous examples of data leak bugs in security-critical libraries such as OpenSSL [99], developers still opt to include such libraries in their TEE application.

Second, by including unvetted third-party code in their TEE application, developers are no longer using TEEs for their originally intended purpose: TEEs are designed to run a small set of trusted application components, which do not need to be run with different privileges. In a large-TCB application, in which different components may require different access privileges to sensitive data, all code running inside an enclave still runs with the same privileges. As such, all third-party code imported into an application running inside a TEE libOS container has unfettered access to all enclave memory leaving the application susceptible to data leaks and corruption.

To make matters worse, most TEE libOSes proposed thus far do not enforce privilege isolation. This crucial security mechanism in traditional OSes distinguishes between user-level and kernel-level processes, ensuring that userspace processes cannot access kernel-level memory. On the other hand, TEE libOSes execute in userspace alongside the application they are running. So, despite running inside a TEE, the libOS's internal management data structures, such as the file descriptor table or the mount table, may be corrupted by vulnerable or malicious third-party code imported by the *application* developer.

To enable developers to reap the benefits of TEEs while making their applications more robust in the face of unvetted third-party libraries, we present Griffin, an in-enclave privilege separation system for large-TCB applications running inside TEEs.

Privilege separation in legacy applications has been the subject of a large body of prior proposals, all with the common goal of enforcing least privilege [109]. Similar to approaches that leverage process isolation to restrict different application components' access to sensitive data or OS resources (e.g., [27, 129, 30]), prior research addressing privilege separation in SGX applications, for instance, has proposed running individual application components in multiple separate SGX processes with communicating enclaves [34, 118, 62].

Yet, these systems either only control access to sensitive data at a per-enclave granularity [34, 62], or they do not protect the TEE libOS itself against application code [34, 118]. Furthermore, this approach requires inter-enclave communication between the different processes to compute on shared data, which incurs significant performance overheads.

In contrast, Griffin subdivides a *single* enclave into multiple isolated memory compartments, each with its own access policy. Griffin then stores developer-specified sensitive in-enclave data objects, such as TLS keys or sensitive datasets, in these compartments. Thus, Griffin can enforce fine-grained least privilege at the granularity of *in-enclave functions* without refactoring the application into multiple TEE processes.

One major challenge in Griffin's single-enclave design is to securely share sensitive enclave data between functions with different access privileges to the same data. Griffin addresses this issue using hardware-assisted memory tagging. Thus, Griffin creates different *memory domains* within an enclave by assigning memory tags to enclave pages.

While the mechanisms we present in this paper apply generally to TEEs, we realize Griffin on top of SGX using Memory Protection Keys (MPK), a memory tagging technique developed at Intel. MPK provides a special hardware register that stores a process' access privileges to each MPK tag, and enforces two types of access privileges to tagged pages (read-only and read-write). Userspace processes may then dynamically adjust the access privileges for different memory tags.

Griffin targets two SGX deployment settings: (1) containerized applications running in a TEE libOS, and (2) SGX-native applications developed using the SGX SDK [69]. Although SGX-native applications typically have a smaller TCB than the libOS setting, developers often still incorporate unvetted third-party libraries into their enclave code for ease of development. Thus, in both settings, Griffin helps developers protect sensitive in-enclave data against leaks or tampering by third-party libraries running in the enclave. In the TEE libOS setting, Griffin can additionally be used to implement OS privilege isolation.

To declare sensitive in-enclave data objects and define those in-enclave functions that are authorized to access them, developers in Griffin specify the sensitive input arguments or return values of third-party library functions executing inside the enclave in a central policy file. At run time, Griffin then assigns each developer-specified data object to an MPK memory domain, and only grants access to those domains when a privileged in-enclave function is executing.

Griffin does not require extensive manual annotations to application source code by the developer, and can be integrated into TEE libOSes to transparently protect the containerized applications they run. We demonstrate the effectiveness of our approach by porting Griffin to the Graphene-SGX libOS [34]. As a preliminary step, our prototype implements privilege isolation in Graphene-SGX to protect the internal libOS data structures, including the file descriptor and mount point tables, against leaks and tampering by untrusted application-level code running on top of the libOS in the enclave.

Although we modified the Graphene-SGX source code to add support for Griffin, applications still run completely unmodified inside the TEE libOS container. To analyze the security properties of Griffin, we first study an adversarial application that mounts a libOS data corruption attack in our prototype, and then examine Griffin's ability to prevent additional hypothetical data leak and corruption attacks within enclave code. We evaluated our Griffin prototype's performance via Graphene-SGX system call microbenchmarks, and find that Griffin imposes acceptable performance and modest memory overheads.

4.2 Background

We instantiate the Griffin memory access control system as an experimental combination of SGX and Memory Protection Keys. This section describes these two techniques, and summarizes the programming framework for SGX applications.

4.2.1 Intel SGX

Intel Software Guard Extensions (SGX) [68, 84] is a trusted execution environment technology that is designed to preserve the confidentiality and integrity of application code and data, even in the face of untrusted or compromised system software.

At its core, SGX provides an isolated memory region within the address space of a userspace process, called an *enclave*. All code and data within the enclave is encrypted and integrity-protected for the entire lifetime of the application. This design divides an application into a trusted and an untrusted context, where the trusted context in the enclave is isolated even from the underlying OS or VMM.

SGX dictates that the enclave must be entered via pre-defined entry points called *ecalls*, and requires that all system calls be made through special trampoline functions that exit the enclave, called *ocalls*. At every ecall and ocall, SGX encrypts or decrypts any memory contents being passed between the contexts, and checks the integrity of all decrypted code. Further, SGX cryptographically computes an *enclave measurement* at application startup,

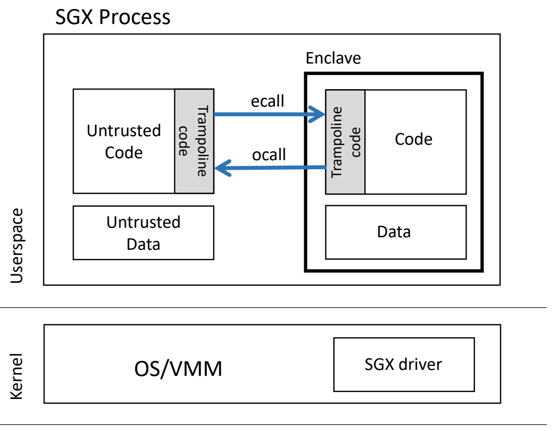

Figure 4.1: Intel SGX application memory layout.

which SGX applications may use to attest to the authenticity of the enclave code. Fig. 4.1 shows the high-level memory layout of an SGX application.

4.2.2 LibOS-based Containers for Intel SGX

SGX-specific library OSes (libOS) such as Graphene-SGX [34], SCONE [17], Panoply [118], and SGXKernel [126], are designed to facilitate the deployment of SGX applications by transparently handling all enclave-to-untrusted transitions necessary to utilize standard OS features. To this end, libOSes provide a trusted shim layer that implements a full userspace-level system call interface essentially creating an in-enclave container that runs unmodified applications on top of the libOS. As an example, Fig. 4.2 shows the system architecture for the Graphene-SGX libOS.

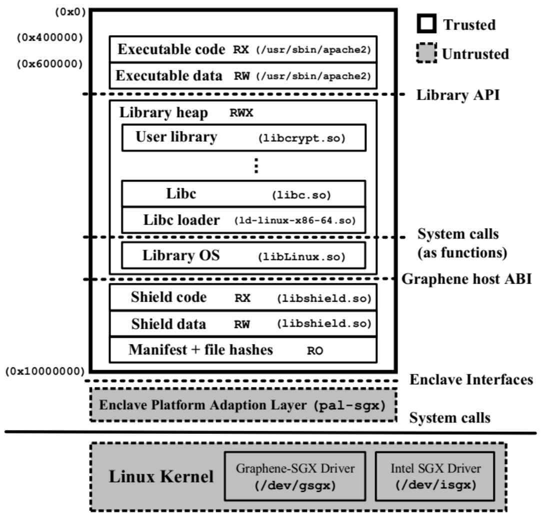
(0x0)
(0x400000)
(0x600000)
Executable code RX (/usr/sbin/apache2)
Executable data RW (/usr/sbin/apache2)
Trusted
Untrusted
Library API
Library heap RWX
User library (libcrypt.so)
Libc (libc.so)
Libc loader (ld-linux-x86-64.so)
System calls (as functions)
Library OS (libLinux.so)
Graphene host ABI
Shield code RX (libshield.so)
Shield data RW (libshield.so)
Manifest + file hashes RO
(0x10000000)
Enclave Interfaces
Enclave Platform Adaption Layer (pal-sgx)
System calls
Linux Kernel
Graphene-SGX Driver (/dev/gsgx)
Intel SGX Driver (/dev/isgx)

Figure 4.2: The Graphene-SGX system architecture (copied from [34]).

We present a prototype of Griffin based on Graphene-SGX, but emphasize that our mechanisms are not specific to Graphene-SGX: our MPK-based memory domain technique may be integrated into other SGX-specific libOSes.

One key feature of Graphene-SGX is its support of dynamically loaded code, requiring only very few additional steps to develop an application for Graphene-SGX. To deploy an application, developers must first specify the resources (e.g., files and iptables-style network rules) required by the application in an application-specific manifest file. Following SGX requirements, the manifest also specifies certain enclave parameters, such as the maximum enclave size and maximum number of threads, which developers may configure based on their application's needs. Developers then run a Linux executable via the Graphene-SGX command-line tool.

4.2.3 Intel SGX-native Software Development

In order to provide SGX's security properties, SGX requires that programmers follow two main development rules when creating an SGX-native application.

First, all enclave code subject to enclave measurement must be static, i.e., built as a statically linked library as part of the whole application. In other words, a major task facing SGX application developers is defining a priori all of the code that runs within the enclave, as well as the ecalls and ocalls for transitioning into and out of the enclave.

To alleviate this effort, the SGX SDK [69] provides the *Edger8r* tool which automatically generates ecall and ocall trampoline functions based on a developer-specified enclave definition file specifying the function signatures for the functions implementing the enclave entry and exit interface. More specifically, Edger8r generates the source and header files for the pre-defined trampoline code, which can then be included as part of the enclave code.

Second, enclave code must be digitally signed to allow SGX to verify the enclave measurement as well as verify the legitimacy of the enclave author. To this end, the SGX SDK also provides the *Sign Tool* that automatically computes the enclave measurement and corresponding signature given the enclave code and the application developer's enclave signing key. Upon enclave startup, SGX can then verify the measurement and signature to confirm the authenticity of the application's enclave.

4.2.4 Memory Protection Keys (MPK)

Memory Protection Keys (MPK) [70] are a recently-developed feature of the Intel x86 architecture, which enables userspace processes to tag each page table entry with a 4-bit *protection key.* Thus, applications create up to 16 isolated *memory domains* within their process address space, and assign individual pages to different domains.

MPK introduces the PKRU, a per-core 32-bit register that stores a bitmap of read-only and read-write access bits for the 16 protection keys. Whenever a process requests access to a page, the memory management unit checks the PKRU to verify that the process has sufficient privileges to access the requested address. To adjust the access permissions to a given domain, an application calls the `WRPKRU` instruction. Though not available on any publicly released CPUs at the time of writing, MPK may also be used to tag SGX enclave pages, providing an additional layer of security to SGX enclave memory.

4.3 System Model and Design Goals

The goal of Griffin is to provide a privilege separation system that protects large-TCB applications running inside a TEE, i.e., containerized applications running inside a TEE libOS as well as large TEE-native applications, against data leaks and corruption by in-enclave third-party code. Griffin combines the security properties of TEEs and hardware-enforced memory tagging to isolate sensitive in-enclave data objects, and only allow privileged in-enclave functions to access these data objects.

4.3.1 Threat Model

As in TEEs, Griffin does not trust any hardware outside the CPU, system software including the OS and any hypervisor, other applications running on the same machine alongside the TEE application, or the untrusted application component running outside the enclave.

However, Griffin assumes a large TCB programming model for TEEs. That is, application developers wish to deploy TEE-protected applications while incorporating third-party libraries that have not been fully vetted (if at all). Thus, while the TEE application developer may have good intentions, Griffin does not trust third-party code the developer imports into her application as part of the enclave code, since it may contain undiscovered data leak vulnerabilities or attacks. Nevertheless, Griffin maintains trust in the TEE libOS container running unmodified applications, as well as in the SGX SDK and its tools used to implement SGX-native applications.

Importantly, Griffin does not aim to mitigate control flow attacks such as buffer overflow or ROP attacks. Such attacks are typically performed as a pathway for an adversary to gain control over the application and execute attack code. Yet, because the adversary's code has already inadvertently been included in the enclave via a third-party library, Griffin assumes that this is sufficient to allow the attacker to moderate the control flow of the enclave. Thus, Griffin's primary goal is to prevent *unauthorized data accesses* that lead to data leaks, corruption or enable other security attacks. Techniques complementary to our approach (e.g., [77, 64]) may be employed to safeguard applications against control flow attacks.

Griffin also does not address side channel or enclave API misuse vulnerabilities such as Iago attacks [35, 62]; countermeasures to these vulnerabilities have been proposed in prior research (e.g., [62, 80, 117, 97, 54]) and are complementary to our work. Much like prior proposals, we also do not address denial-of-service attacks.

4.3.2 Security Properties

Our design for Griffin provides the following three security properties.

P1: Least Privilege. An in-enclave function may only access those sensitive enclave data objects that this function needs to provide its expected functionality. Griffin achieves such intra-process least privilege by placing developer-specified sensitive enclave data objects in isolated memory compartments, and enforcing a *default-deny* access policy to these compartments at the granularity of individual functions running inside the enclave.

P2: Single Enclave Isolation. Strong isolation of sensitive in-enclave data does not require partitioning a TEE application into multiple enclaves. Griffin relies on a hardware-assisted memory isolation technique capable of controlling access to individual enclave pages, enabling the creation of isolated memory compartments *within* a single enclave.

P3: Secure Data Sharing. Multiple in-enclave functions may need to operate on the same sensitive data object. To enable data sharing between functions that may have different access privileges within a single enclave, Griffin's memory compartments support dynamic access privileges. That is, Griffin enforces access permissions to a given sensitive data object based on the function requesting access.

Non-goals. Griffin automatically controls access to sensitive data inside a TEE at the level of in-enclave functions. However, Griffin does not aim to provide automated application code partitioning as in Glamdring [80]. Such execution isolation is orthogonal to our approach, and could be used in conjunction with Griffin to further reduce the TCB of TEE applications. We assume that the application developer has partitioned her application into trusted and untrusted components either manually or using an automated technique.

Griffin also does not ensure the *correctness* of the sensitive in-enclave data it protects. While providing these stronger security guarantees ensure the integrity of sensitive enclave inputs and help prevent additional data leaks via buggy or malicious third-party code at run time, formally verifying the implementation of enclave code, as well as the provenance of enclave inputs, is orthogonal to Griffin's goals.

4.4 Griffin Design

Our design for Griffin provides isolation and access control for sensitive in-enclave data objects in the face of untrusted third-party enclave code, without partitioning the TEE application into multiple enclaves. In §3.5, we describe how Griffin does not require extensive manual annotations to application source code by the developer, and how libOSes can integrate Griffin to transparently protect the containerized applications they run.

Much as in prior research [89, 113], Griffin relies on the application developer's understanding of the libraries that they import, and on their coarse-grained expectations of how these libraries access sensitive data. Based on this information, Griffin requires developers to specify access privileges for individual in-enclave functions to specific sensitive data objects.

At enclave startup, Griffin maps these data objects to in-enclave memory domains and partitions the enclave into corresponding compartments based on the developer's policy §4.4.1. Then at run time, Griffin monitors the enclave execution, and dynamically adjusts the enclave's access to the memory domains according to the currently executing function §4.4.2. Figure 4.3 shows the main components of the Griffin system.

Griffin uses memory domains to isolate sensitive data objects within an enclave address space. Each domain is then mapped to its own MPK tag to control access to the domain at run time. To specify in which domain Griffin will place a given data object, developers may also assign a unique label to each domain in their policy.

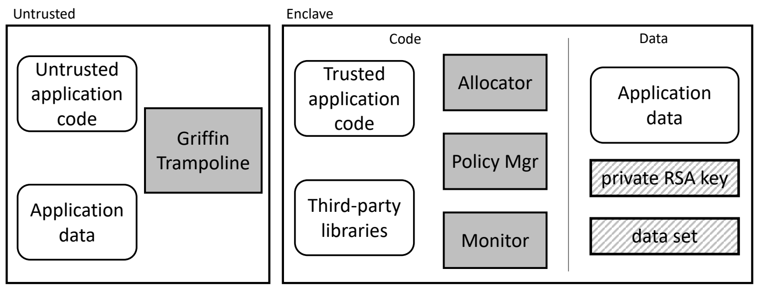

Figure 4.3: Overview of Griffin, which isolates sensitive in-enclave data in memory domains (striped boxes), and controls access to these domains with three key components (gray boxes).

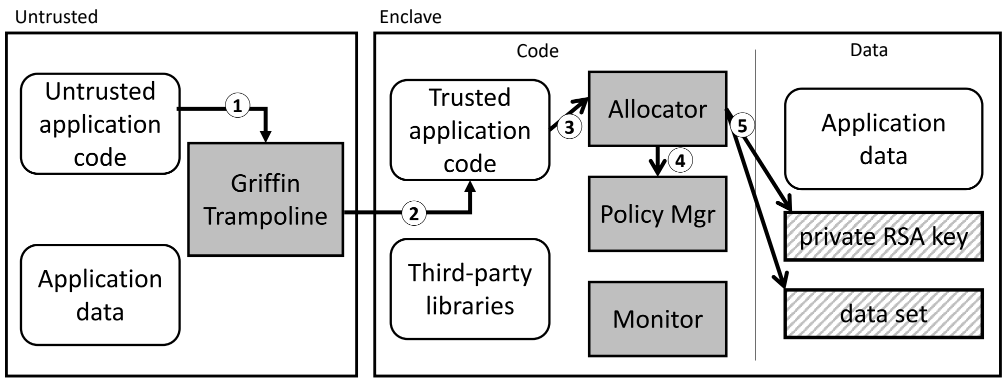

Figure 4.4: The steps that Griffin takes at enclave startup to partition an enclave into memory domains.

4.4.1 Enclave Compartmentalization

The Griffin Policy Manager maintains an access control list (ACL) based on the developer-supplied data access policy. As we show in Fig. 4.4, at enclave startup, the Griffin Allocator partitions the enclave into memory domains based on the ACL, and is then responsible for allocating sensitive data objects in their corresponding domains at run time.

Since Griffin aims to protect a wide variety of sensitive data objects, memory domains support flexible data object sizes. To this end, each memory domain consists of a *pool* of enclave pages all tagged with the same MPK tag. Notably, developers need not know the exact size of their sensitive data objects in order to isolate them in a domain since Griffin allows developers to dynamically allocate domain memory at run time.

This mechanism allows Griffin to support arbitrary and/or variable data object sizes. For added flexibility, developers may configure the maximum number of pages per pool based on an *estimate* of the maximum memory consumption of their specified data objects.

It is important to note that there is a trade-off between the granularity of data object isolation (i.e., security) and an application's memory footprint. Recall from §4.2.4 that changes to the PKRU affect all memory pages with the same MPK tag. This means that, on the one hand, if an application's memory domains contain only single, small data objects, most of the reserved enclave pages in the domains' pools will remain unused throughout the lifecycle of the application inflating the memory footprint of the application.

However, this policy guarantees that every data object is only accessible to those functions that have sufficient privileges to access the corresponding memory domains. On the other hand, a developer may use the reserved page pool more efficiently by placing multiple data objects in the same memory domain, at the risk of potentially allowing a function that *should* only access one of those data objects to leak another sensitive data object in the same domain.

4.4.2 Dynamic Sandboxed Execution

MPK restricts access to memory pages at the CPU core level meaning that *all* code running on the same core has access to all tagged pages whose PKRU access bits make those pages accessible. However, to enforce least privilege, Griffin cannot allow enclave functions with insufficient privileges to access unauthorized data objects.

To protect sensitive data objects in face of leaks or tampering by an untrusted third-party library at run time, Griffin ensures that only those memory domains to which the currently executing in-enclave function has privileges are accessible, and denies access to all other domains. Specifically, upon entering a privileged in-enclave function, the Griffin Monitor queries the Policy Manager for the given function's data object access rules, and encapsulates the function in an execution sandbox by dynamically adjusting the access bits for the MPK tags associated with the corresponding memory domains (see Fig. 4.5).

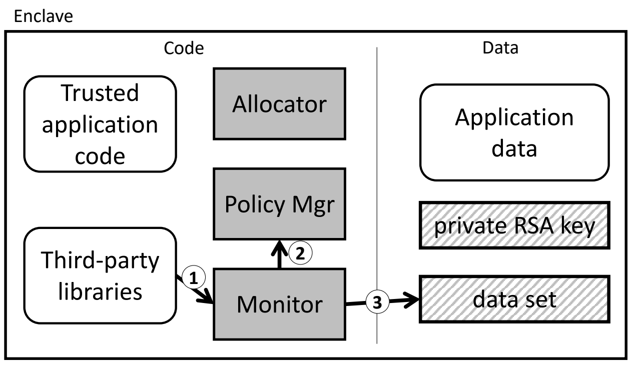

Figure 4.5: The steps that Griffin takes when entering an in-enclave function that may require access to domain-protected data.

After returning from the sandboxed function, and before exiting the execution sandbox, the Griffin Monitor then once again restricts access to all memory domains. While this step may be avoided if the next enclave function to be executed also has access to some or all of the accessible domains, Griffin takes a conservative approach and makes no assumptions about the control flow of applications.

4.4.3 Versus MPX and MKTME

We also considered two additional Intel hardware-assisted memory isolation techniques that have been announced recently for Griffin's memory access control mechanism: MKTME [66] and MPX [67]. Indeed, prior research has proposed using MPX for in-enclave data isolation in library OSes [116]. However, we ultimately found these two techniques to be inadequate for building a fine-grained, single enclave memory access control mechanism.

MKTME (or Multi-Key Total Memory Encryption) provides a number of principals each with a unique cryptographic key for encrypting specific memory regions within a single address space. At first glance, this technique seems adequate for implementing memory access control since the possession of cryptographic key for a specific memory region implies access permissions to that memory region.

However, MKTME does not support dynamic access privileges to a given memory region, which is a significant barrier for secure data sharing. Further, MKTME relies on the OS for operations such as encryption key provisioning, which would leave SGX applications vulnerable to compromised system software.

MPX (or Memory Protection Extensions), on the other hand, performs memory boundary checks in hardware, allowing developers to define boundaries around different code or data portions of an application to enforce privilege separation within the same address space. While MPX operations are performed solely in hardware avoiding any OS interactions (as in MKTME), MPX also does not easily support secure data sharing since access privileges to different bounded memory regions are static.

4.4.4 MPK Limitations

MPK has a two important drawbacks that may leave Griffin-protected applications vulnerable to leaks. First, contrary to the SGX threat model, in which no software outside of the enclave is trusted, even the underlying system software, MPK relies on the OS to provision MPK tags and to assign tags to given memory pages (similar to MKTME). Thus, an attacker may gain unauthorized access to MPK-protected memory via a compromised OS.

Second, MPK does not control *who* performs the PKRU access bits adjustments (i.e., what userspace process calls `WRPKRU`), which may enable an adversary running a process on the same core as the Griffin-protected SGX application to make unauthorized changes to the PKRU, leaving sensitive data objects accessible to untrusted code within the enclave. Addressing these limitations is the subject of future research.

4.5 Implementation

We implement Griffin as a userspace library that can be integrated into SGX-specific libOSes and SGX-native applications in a handful of steps, and we demonstrate Griffin's practicality and benefits with our Griffin-protected Graphene-SGX libOS prototype.

4.5.1 Griffin API

To facilitate adoption and usability, our userspace library for Griffin exposes a small programming interface which abstracts away the use of MPK. Importantly, the execution sandboxing API calls can be automatically generated in both of Griffin's targeted deployment settings. For developers writing SGX-native applications, we envision providing a modified version of the *Edger8r* tool to minimize the number of manual changes to application source code.

Table 4.1: The Griffin API. In a Graphene-SGX container deployment setting, the API is used by the *libOS* developers. For an SGX-native deployment setting, the * denotes API calls that can be automatically generated based on the developer's policy.

Initialization/teardown
`int griffin_init(void)`
`void griffin_teardown(void)`
Data object management*
`int griffin_check_input_size(const char *obj_label, size_t size)`
`int griffin_check_output_size(const char *obj_label, size_t size)`
`int griffin_copy_from_untrusted(const char *obj_label, void *untrusted_buf)`
`int griffin_copy_to_untrusted(const char *obj_label, void *untrusted_buf)`
Domain memory management*
`void *griffin_malloc(const char *domain_label, size_t size)`
`void griffin_free(const char *domain_label, void *addr)`
Execution sandboxing*
`int griffin_grant_data_access(const char *func_name)`
`void griffin_revoke_data_access(const char *func_name)`

Further, to allow developers of containerized applications (i.e., those running inside a libOS) to protect sensitive data via Griffin, we could provide a tool for libOS developers to transparently enforce the application developer's data access policy, and automate intra-process execution sandboxing for application-level library code.

We also provide a policy generation tool that automatically populates header files with the internal representation of the application's data object ACL (see §4.5.4). To maintain the integrity of the developer's data access policy, we do not provide any policy-related API calls; the internal representation of the policy is only accessible within the Griffin code. Table 4.1 shows Griffin's API.

Initialization and teardown. In order to enable Griffin's run-time memory access control, developers must use the initialization and teardown API. Notably, these functions should only be called once each, and may be called either from the untrusted application context or from within the enclave. Given the data object ACL generated at enclave build time, `griffin_init()` is responsible for pre-allocating the enclave pages reserved for memory domains, and for provisioning and mapping MPK tags to memory domains. Correspondingly, `griffin_teardown()` releases provisioned MPK tags with the OS, and frees the memory domain enclave pages. These two API calls are the only Griffin functionality that libOS developers and SGX-native application developers *must* manually incorporate into their application source code.

Data object management. SGX-native application developers may wish to pass data into the enclave and treat it as sensitive data, or return a data object to the untrusted context

Table 4.2: Supported data access semantics and their corresponding policy rule specification.

Access semantics	Object specification
no writable objects	$O_O = \emptyset$
all writable objects	$O_I = \emptyset$
blanket access to domain	`#<domain label>:`
skip object size verification	`<object label>#<domain label>:`

as the result of an enclave computation. To support passing sensitive data objects during ecalls and ocalls, Griffin provides developers with an API for copying a given sensitive data object into or out of the enclave. As part of passing sensitive data objects between contexts, Griffin also allows developers to verify the size of the data objects. This API may only be called from within an enclave. To ease adoption, these API calls can be easily integrated into the SGX SDK's *Edger8r* edge routine generation tool.

Domain memory management. As we describe in §4.4.1, applications in Griffin may allocate data in memory domains dynamically at run time. Griffin's memory management API allows libOS and SGX-native application developers to either replace or complement existing malloc calls inside the enclave with domain-specific memory allocations. As with the data object management API, this functionality can be integrated into the *Edger8r* tool to facilitate the adoption process for SGX-native applications.

Execution sandboxing. While developers must explicitly specify each sensitive data object that a privileged in-enclave function may access in their policy, Griffin transparently maintains all data object-to-domain mappings. Developers then use Griffin's execution sandboxing API to specify the boundaries of a function sandbox by encapsulating privileged function calls between a `griffin_grant_data_access()` call at the entry of a privileged function, and the corresponding `griffin_revoke_data_access()` call after returning. In §4.5.4, we describe how Griffin can help to automate this process for application developers in both deployment settings.

4.5.2 Policy Specification

Griffin requires application developers to declare sensitive data objects and specify corresponding enclave function-level access rules in a static policy file. In addition, libOS developers may also generate a Griffin policy to protect internal libOS data structures that is included in the libOS core (see §4.5.3). Every access rule takes the form:

O_I > $func$ > O_O

where $func$ is the name of the in-enclave function affected by the given rule. O_I specifies those sensitive data objects to which $func$ has read-only access (i.e., function inputs), and O_O specifies those data objects to which $func$ has read-write access (e.g., function return values, or pass-by-reference arguments).

In an access rule, O_I and O_O are written as comma-separated lists of *object specifications* formatted as

`<object label>#<domain label>:<object size>`.
To allow developers to tailor their policies to their application's security and performance requirements, Griffin supports flexible data access semantics, detailed in Table 4.2.

Additionally, Griffin does not require a 1:1 correspondence between data objects and memory domains. That is, multiple individual data objects may be placed into the same domain. Such access rules may be beneficial in cases in which a number of different in-enclave functions require access to the same individual data objects.

4.5.3 Privilege Isolation for Graphene-SGX

As described in §4.2.2, libOSes such as Graphene-SGX [34] provide application developers with a mechanism for reaping the benefits of SGX, while running unmodified applications. However, Graphene-SGX's trusted shim layer performing privileged operations (i.e., system calls) runs with the same privileges as the application binary and any third-party libraries imported into the application. This lack of privilege isolation poses a threat to the libOS and any applications it runs because vulnerabilities or attacks in untrusted third-party code may corrupt or leak sensitive internal libOS data.

As a preliminary step to demonstrate Griffin's ability to protect large-TCB applications against third-party code, we have built a Griffin-enabled prototype of the Graphene-SGX libOS implementing a privilege isolation mechanism. Our current prototype enforces privilege isolation by placing all file system-related data structures into memory domains, and by only granting access to this data within relevant system calls. More specifically, we declare two memory domains: *handle_dom* contains all file descriptor-specific metadata, and *fs_dom* contains all file system management metadata (e.g., the mount table).

Our decision to create these two domains was largely based on Graphene-SGX's implementation of the file system interface, which treats these two types of metadata separately employing a separate memory manager for each. As such, we were able to replace Graphene-SGX's internal memory allocation calls in these memory managers with Griffin domain memory management API calls for the corresponding memory domains.

We currently declare data access rules for 32 libOS system calls, for which we automatically generate the Griffin policy code with our policy generator. However, porting Griffin to Graphene-SGX required us to manually implement the sandboxing wrapper functions for the 32 system calls since the libOS employs a custom interface to SGX; this interface is unfortunately not compatible with the official SGX SDK, hindering us from automatically generating the Griffin system call wrappers with the Edger8r tool. Furthermore, to comply with Graphene-SGX's system ABI and make Griffin API calls from the Library OS layer, we needed to create additional wrapper functions for Griffin's API in the Host layer.

Nevertheless, our instrumentation only affects the Graphene-SGX libOS itself, so application developers who do not wish to provide additional application-level data protections can still remain agnostic to Griffin (as well as SGX) by running unmodified applications on top of our Griffin-enabled Graphene-SGX prototype.

4.5.4 Griffin Code Generation

Policy generator. Griffin protects the developer's policy against tampering by untrusted code by leveraging SGX's measurement of static enclave code. To avoid having developers manually create an internal representation of their policy, we provide a *policy parser* that automatically populates Griffin's policy data structures based on the developer's policy, and generates a header file containing the internal representation of the data object ACL.

Developers run the policy generator as an additional step in their application build process, and include the generated ACL header file in the libOS or application source code. The Griffin API can then access these policy data structures directly at run time.

Finally, the policy generator tool allows libOS and SGX-native application developers to configure the number of enclave pages reserved for a memory domain. If no domain pool size is specified, the default of four 4-KB pages is used.

Griffin Edger8r tool. Since most SGX-native applications do not compute on static sensitive data, developers must often pass sensitive data objects between the untrusted context and the enclave at run time. To examine how much we can reduce the burden on developers and to help ensure that Griffin-protected SGX-native applications fully protect sensitive data objects, we built an experimental Griffin-aware Edger8r tool that automatically inserts data object and domain memory allocation calls in the generated trampoline code based on the developer's Griffin policy.

In addition, Griffin can bootstrap the vanilla Edger8r's functionality to create trampoline code to create application-specific wrappers around privileged in-enclave functions to create the function execution sandboxes. These generated sandboxing wrappers include calls to Griffin's execution sandboxing API surrounding the actual call to the privileged function. Developers would then only be required to replace the calls to the original function with calls to the sandboxing wrapper function to ensure that the relevant sensitive data objects may be accessed within the scope of the sandboxed function. Any "un-sandboxed" calls to the original function would proceed but attempts to access sensitive data objects will be blocked by Griffin's Monitor.

Griffin for containerized applications. While not currently implemented in our Graphene-SGX prototype, libOS developers may wish to provide support for Griffin to *application* developers. To keep the burden on the application developers minimal, they would only be required to specify a Griffin policy specifying the privileged third-party functions their application calls and the sensitive data objects these functions may access. Griffin would provide a tool for automatically generating the ACL header file which developers could include in their application's source code. We then envision Graphene-SGX transparently parsing the application's data object ACL header file, and allocating the developer-specified data objects in their respective memory domains. To create application-specific dynamic execution sandboxes automatically at run time, we could leverage Graphene-SGX's special linker to create special wrappers around the privileged functions.

4.6 Evaluation

We evaluate Griffin's security properties by performing a case study of an adversarial application that we run in our prototype, and present an analysis of additional hypothetical vulnerabilities in enclave code. To evaluate the performance and memory overheads Griffin imposes on the Graphene-SGX libOS, we ran microbenchmarks to understand how the underlying Griffin operations affect our measurements.

Our tests were performed on a machine with an experimental Intel CPU with two 224 GB Intel SSDs, running Ubuntu 17.04 on Linux Kernel 4.10.0-42-generic. Intel CPUs that feature both SGX and MPK are unavailable commercially at the time of writing. Further, the policy for our prototype contains access rules for 32 libOS system calls, and is configured to provision a maximum of four 4-KB memory pages per domain.

4.6.1 Security Analysis

To evaluate Griffin's security properties, we first do a case study on an application that attempts to tamper with Graphene-SGX's file descriptor table. We developed this adversarial application in our laboratory setting, demonstrating the feasibility of such attacks without Griffin's protections, and Griffin's broader applicability to similar attacks. Furthermore, we also analyze additional hypothetical vulnerabilities in enclave code.

Case study: Descriptor Table Corruption. To demonstrate Griffin's ability to mitigate unauthorized accesses, we examine the instance of file descriptor table corruption in a libOS-based SGX application. If successful, such attacks can be especially detrimental since they may enable a wide range of malicious application behaviors.

In this case study, the adversarial application imports our specially crafted malicious library which attempts to corrupt Graphene-SGX's file descriptor table (or FD table). We ran this adversarial application in vanilla Graphene-SGX and our Griffin-enabled Graphene-SGX prototype, and found that the attack succeeds in vanilla Graphene-SGX, where Griffin prevents this attack via a segfault.

This attack is possible in vanilla Graphene-SGX because the libOS itself and the adversarial application run in the same address space. Recall from §4.5.3 that our prototype places the FD table in the *handle_dom* memory domain, which is only accessible during those libOS system calls specified in our prototype's data access policy. In other words, since our crafted library does not make system calls that Griffin executes in a sandbox, but rather attempts to access the FD table via a reverse-engineered code path that does not run inside a dynamic sandbox, access to the *handle_dom* is never granted to the crafted library.

While the attack in this case study may seem contrived at first glance, we believe that file descriptor table corruption demonstrates a broader vulnerability class that allows third-party code to abuse its access to in-enclave data. Griffin mitigates this threat to TEE applications, which arises specifically because developers include untrusted third-party code.

In addition, this case study shows that Griffin can prevent a more insidious class of attacks capable of circumventing more traditional access control methods such as system call interposition: since our studied attack does not require direct system calls to access a

Table 4.3: Hypothetical vulnerabilities in enclave code, mitigation strategies, and whether Griffin currently implements them.

Vuln Type	Mitigation	Implemented?
confused deputy attack	execution sandbox	Y
WRPKRU misuse	static analysis	N
MPK OS-level API misuse	integration with libmpk [100]	N

sensitive data object, interposition mechanisms that check whether the application is only accessing authorized OS resources would not even be invoked.

Hypothetical vulnerabilities. We consider a range of vulnerabilities that untrusted third-party code running inside an enclave may hypothetically exploit in order to leak sensitive in-enclave data to the untrusted application component. Table 4.3 summarizes three hypothetical vulnerabilities, and the specific Griffin mechanism that mitigates each vulnerability.

Griffin allows developers to grant specific in-enclave functions privileges to access certain sensitive data objects. However, an untrusted third-party function running inside the enclave may attempt to escalate its own privileges by calling a more privileged function in order to gain unauthorized access to sensitive data objects. Griffin prevents such *confused deputy attacks* [55] via its dynamic execution sandbox mechanism (see §4.4.2).

Untrusted in-enclave code may attempt to manipulate the value of the PKRU by including direct `WRPKRU` instructions within its code. Since MPK does not currently verify the origin of `WRPKRU` instructions in a userspace process, one viable mitigation strategy that Griffin could employ is static analysis and binary instrumentation in order to detect and replace errant `WRPKRU` instructions in non-Griffin enclave code (similarly as in ERIM [128]).

Finally, since MPK relies on the OS for provisioning protection keys, in-enclave code may attempt to gain authorized access to MPK-protected memory or corrupt the PKRU by directly making `pkey_mprotect` or `pkey_set` system calls via an ocall. Concurrent research on libmpk [100] proposes techniques for mitigating these issues, and may be integrated with Griffin as well.

4.6.2 Performance Microbenchmarks

Execution time overhead. To analyze the impact of Griffin on the performance of the Graphene-SGX libOS, we took microbenchmarks of the `open`, `stat`, `fstat`, `mmap`, and `close` system calls. Recall from §3.5 that our Griffin-enabled prototype of Graphene-SGX isolates the libOS internal file descriptor and file system management data structures (e.g. mount table) into two memory domains. Thus, we chose to benchmark these five system calls since they require access either only to the file descriptor domain *handle_dom*, the file system management domain *fs_dom*, or both, and sought to determine how accessing the different memory domains affects performance.

We used the `lat_syscall` benchmark of LMbench 2.5 [85], which stress tests six system calls; for each LMbench experiment, we then measured the amount of execution time of each

Table 4.4: Mean percentage of execution time spent performing Griffin operations for five libOS syscalls.

	% exec time in Griffin	**accessed memdom(s)**
`open`	6.4	handle, fs
`close`	49.1	handle
`stat`	49.9	fs
`fstat`	50.1	handle, fs
`mmap`	0.8	handle

Table 4.5: Peak memory usage for internal Griffin data structures for each domain as well as the total Graphene-SGX peak memory usage in our prototype.

	memory usage (in bytes)
handle_dom	98
fs_dom	1030
Total	1200

our target system calls spent performing Griffin operations. Table 4.4 shows the mean percentage of execution time that Griffin operations comprise in our five benchmarked syscalls as well as the Griffin memory domain that each syscall accesses internally.

We observe two groups of system calls: the first group, `mmap` and `open`, spends only a small portion of execution time (at most 6.4%) performing Griffin operations, while the second group of syscalls spends about 50% of the execution time performing Griffin operations. This result is perhaps not entirely unsurprising given that the primary purpose of the system calls in the second group is to access the file system metadata, while `open` and `mmap` perform a much larger number of tasks beyond manipulating the internal file system data structures.

Memory overhead. Our evaluation additionally sought to quantify the memory overhead that Griffin imposes on Graphene-SGX. Specifically, we measure the additional memory required to maintain the application ACL and internal memory domain management data structures for each domain. Table 4.5 shows the median peak memory consumption in bytes of Griffin's internal data structures for our Graphene-SGX prototype as a whole as well as each memory domain during our syscall microbenchmarks above.

We find that Griffin's memory overhead is very modest requiring only an additional 1.2 KB of memory for the entire Graphene-SGX libOS, and a mean of 0.6 KB per memory domain. Note that our configuration provisions far more memory per domain than is required.

4.7 Related Work

In-SGX memory protection. A small number of prior proposals have sought to enhance memory protection *within* an SGX enclave. SGXBounds [77] uses tagged pointers for efficient bounds-checking within an SGX enclave. SGX-Shield [114] provides a new ASLR scheme to protect SGX applications against memory corruption attacks inside the enclave. T-SGX [117] leverages the hardware transactional memory provided by Intel TSX to help prevent controlled-channel attacks in an SGX enclave. Multi-domain SFI [116] subdivides an SGX enclave into multiple memory domains using MPX's hardware-enforced memory bounds checking to implement privilege isolation and multi-process library OSes within a single enclave.

All of these prior proposals (except Multi-domain SFI) aim to prevent a particular class of memory attacks. In contrast, Griffin operates at the granularity of application-level data objects to help developers prevent sensitive data leaks in a more intuitive fashion. Multi-domain SFI is closest to our approach, but we believe Griffin's use of MPK, a technique designed specifically for memory access control, provides a more adequate solution for implementing in-enclave privilege separation with secure data sharing.

Intel hardware memory protection in legacy applications. Numerous prior proposals have leveraged novel techniques available in Intel processors to help protect sensitive application data within a single address space.

ERIM [128], MemSentry-PKU [75] and Janus [57] use MPK to partition a legacy application's address space into two or more domains to isolate sensitive application data. Griffin builds on this prior research generalizing the use of MPK applies this mechanism to SGX applications, leveraging MPK to partition an SGX enclave and isolate sensitive in-enclave data objects in multiple separate domains.

Systems such as Dune [22], MemSentry [75], and Janus [57] rely on Intel VT-x virtualization hardware to create intra-process isolated compartments. Unlike Griffin, the main goal of these approaches is to provide *execution* isolation for running different application components in separate address spaces. Griffin's main goal is to protect sensitive application data shared between different application components by controlling access to sensitive data objects at run time.

Non-Intel hardware memory isolation. Other architectures, most notably ARM, also provide techniques designed for finer-grained memory access control. For instance, ARMlock [145], Shreds [36], and FlexDroid [113] leverage ARM memory domains, a technique very similar to MPK, to allow developers to create a number of isolated execution regions with associated private memory compartments.

Process isolation. A large number of prior proposals have leveraged OS primitives to restrict application components in legacy and mobile applications. Systems such as Wedge [26], Privtrans [30], Passe [27], CodeJail [137], AdSplit [115] and BreakApp [129] partition an application into multiple processes to run unprivileged components in isolated address spaces. Ryoan [62] partitions an SGX application into separate SGX processes as a means to isolate different application components.

While such process-based isolation provides strong memory protections, these approaches require significant development efforts to re-architect a monolithic application for a multi-process model. Griffin's single-process design, on the other hand, provides a more practical approach for SGX application developers that does not require major application refactoring, and still provides strong hardware-assisted memory isolation.

Intra-process isolation. Arbiter [132], SMV [59], Light-weight Contexts [82], and Pyronia [89] partition a single process address space using multiple page tables in order to control access to different memory compartments at intra-process granularities (e.g., thread-level in Arbiter or SMV). Griffin borrows many concepts from these intra-process isolation proposals, but relies on more efficient hardware-based techniques that are more adequate for use in SGX applications.

4.8 Griffin Summary

We have presented Griffin, a privilege separation system for large-TCB applications running in trusted execution environments. Griffin combines the security of TEEs with memory tagging to prevent sensitive data leaks and corruption by untrusted third-party code imported into containerized and TEE-native applications. To avoid partitioning an enclave into multiple TEE processes, Griffin isolates sensitive data objects within a single enclave address space. As such, Griffin supports data sharing between different in-enclave functions, while enforcing intra-process least privilege data access policies.

We have implemented Griffin as a userspace API for large-TCB SGX applications that uses MPK for enclave memory tagging. To demonstrate Griffin's properties, we implement OS privilege isolation in the Graphene-SGX library OS. Our evaluation of our Griffin-enabled Graphene-SGX prototype imposes a very modest memory overhead and can protect containerized applications against unauthorized accesses to sensitive in-enclave data.

Chapter 5

Conclusion

Compute platforms have evolved immensely since the Principle of Least Privilege was first introduced in the mid 1970s. In today's digital world, computers are seemingly ubiquitous, and we rely on software applications for even the most privacy-sensitive tasks. Unlike earlier monolithic applications that were developed primarily by a single entity, today's applications are more complex making heavy use of third-party code.

This dissertation examined the security impact of third-party library use in today's applications, and existing solutions for enforcing the Principle of Least Privilege in the face of untrusted application components. Given the high sensitivity of the data today's applications collect and process, we find that existing least privilege approaches that require developers to make major modifications to their applications, and are largely not generally applicable in broader application domains, are insufficient for protecting against the threat of untrusted third-party code.

To address this crucial issue, we proposed intra-process least privilege, a design principle for protecting applications specifically against imported third-party code without requiring major efforts from application developers. At its core, intra-process least privilege enforces access policies to sensitive data objects at the granularity of programming language functions, and partitions a single process address space into least-privilege compartments to isolate sensitive in-memory data.

Next, we explored how intra-process least privilege enables developers to protect their applications in two emerging application domains: the Internet of Things and trusted execution environments. We described how the Pyronia privilege separation system protects IoT applications written in high-level languages against OS resource-based data leak vulnerabilities in third-party libraries. Further, the Griffin privilege separation system for large-TCB TEE applications demonstrates how intra-process least privilege can enhance the security of TEEs. Both systems impose acceptable execution time overheads, and modest memory overheads in existing applications.

Future Work. Pyronia and Griffin represent only the first steps towards a usable privilege separation approach that today's application developers may adopt. We envision several avenues for future research.

Pyronia targets IoT applications written in high-level languages such as Python or JavaScript. These are very popular programming languages in other domains such as server applications, and they face the same threats from potentially vulnerable third-party libraries. One direction for future work could investigate how Pyronia specifically, and intra-process least privilege more generally, can be applied in other application domains.

Our work thus far has focused on isolating sensitive *data*, but control-flow attacks such as ROP attacks pose a serious challenge to intra-process memory protection mechanisms. Another interesting future direction could explore how to extend intra-process protections to executable memory.

Finally, as we discussed in §3.8, there is much room for improving upon policy specification. One of the primary reasons prior least privilege proposals have seen little real-world adoption is due to the high overhead to adopt these proposals. As such, we recognize the importance of minimizing the burden on developers by learning more about developers needs to build tools that can assist them to make informed decisions and make the policy specification more intuitive without weakening application security.

Final Remarks. The threats posed by third-party code are not new, or IoT- or TEE-specific. However, given the rapid proliferation of IoT devices, and the increasing use of TEEs for sensitive applications in the cloud, we believe it is crucial to make least privilege and isolation practical. We hope that the work presented in this dissertation will raise awareness about the risks of using unvetted third-party code, and ultimately drive further development of usable security tools for application developers.

Bibliography

[1] AWS IoT Device SDK for Python. https://github.com/aws/aws-iot-device-sdk-python.

[2] CVE-2019-9948. Retrieved Aug. 2 2019, from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-9948.

[3] MQTT.org. Retrieved Feb. 9, 2019, from https://mqtt.org.

[4] paho-mqtt. Python Package Index (PyPI), https://pypi.org/project/paho-mqtt/.

[5] Security and Privacy Day. National Security Institute, https://nationalsecurityinstitute.org/spday/, Oct 2017.

[6] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky. You Get Where You're Looking for: The Impact of Information Sources on Code Security. In *Proc. IEEE Symposium on Security and Privacy*, May 2016.

[7] Yasemin Acar, Sascha Fahl, and Michelle L. Mazurek. You are Not Your Developer, Either: A Research Agenda for Usable Security and Privacy Research Beyond End Users. In *Proc. IEEE Cybersecurity Development*, pages 3–8, 11 2016.

[8] Amazon Web Services, Inc. Amazon Web Services (AWS) - Cloud Computing Services. Retrieved Aug 2 2019, from =https://aws.amazon.com.

[9] Amazon Web Services, Inc. AWS IoT. Retrieved Feb. 9, 2019, from https://aws.amazon.com/iot/.

[10] Amazon Web Services, Inc. AWS IoT Plant Watering Sample. Retrieved Feb. 9, 2019, from https://docs.aws.amazon.com/iot/latest/developerguide/iot-plant-watering.html.

[11] Amazon Web Services, Inc. Security and identity for AWS IoT. AWS IoT Devloper Guide.

[12] Amazon.com, Inc. Alexa Voice Service. Retrieved Feb. 9, 2019, from https://developer.amazon.com/alexa-voice-service.

[13] AppArmor maintainers. AppArmor History. Accessed Aug 1, 2019, https://gitlab.com/apparmor/apparmor/wikis/AppArmor_History.

[14] AppArmor maintainers. AppArmor security project wiki. Accessed Dec. 30, 2018, https://gitlab.com/apparmor/apparmor/wikis/home/.

[15] ARM Limited. Domains. Retrieved Feb. 12, 2019, from http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0211k/Babjdffh.html.

[16] Arm LimitedI. Introducing Arm TrustZone. https://developer.arm.com/ip-products/security-ip/trustzone. Accessed 2 Aug 2019.

[17] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *OSDI*, 2016.

[18] Elias Athanasopoulos, Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. Nacldroid: Native code isolation for android applications. In Ioannis Askoxylakis, Sotiris Ioannidis, Sokratis Katsikas, and Catherine Meadows, editors, *European Symposium on Research in Computer Security (ESORICS)*, 2016.

[19] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. Boxify: Full-fledged app sandboxing for stock android. In *USENIX Security Symposium*, 2015.

[20] R. Balebako and L. Cranor. Improving App Privacy: Nudging App Developers to Protect User Privacy. In *Proc. IEEE Symposium on Security and Privacy*, July 2014.

[21] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *OSDI*, 2014.

[22] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *Proc. OSDI*, 2012.

[23] Benjamin Cabé. IoT Developer Survey Results 2018. https://www.slideshare.net/kartben/iot-developer-survey-2018, 2018. Accessed 21 Dec 2018.

[24] Antonio Bianchi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. MAPbox: Using parameterized behavior classes to confine untrusted applications. In *USENIX Security Symposium*, 2000.

[25] Antonio Bianchi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. NJAS: Sandboxing unmodified applications in non-rooted devices running stock android. In *Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2015.

[26] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *NSDI*, 2008.

[27] Aaron Blankstein and Michael J. Freedman. Automating Isolation and Least Privilege in Web Services. In *IEEE Symposium on Security and Privacy*, 2014.

[28] Theodore Book, Adam Pridgen, and Dan S. Wallach. Longitudinal Analysis of Android Ad Library Permissions. *arXiv e-prints*, page arXiv:1303.0857, March 2013.

[29] Julie Bort. For The First Time, Hackers Have Used A Refrigerator To Attack Businesses. Business Insider, Jan. 2014.

[30] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium*, 2004.

[31] Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *USENIX Security Symposium*, 2013.

[32] Justin Cappos, Armon Dadgar, Jeff Rasley, Justin Samuel, Ivan Beschastnikh, Cosmin Barsan, Arvind Krishnamurthy, and Thomas Anderson. Retaining Sandbox Containment Despite Bugs in Privileged Memory-safe Code. In *17th ACM Conference on Computer and Communications Security (CCS)*, 2010.

[33] Catalin Cimpanu. Twelve malicious Python libraries found and removed from PyPI. https://www.zdnet.com/article/twelve-malicious-python-libraries-found-and-removed-from-pypi/.

[34] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX ATC*, 2017.

[35] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. In *Proc. ASPLOS*, 2013.

[36] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-Grained Execution Units with Private Memory. In *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[37] Winnie Cheng, Dan R. K. Ports, David Schultz, Victoria Popic, Aaron Blankstein, James Cowling, Dorothy Curtis, Liuba Shrira, and Barbara Liskov. Abstractions for usable information flow control in aeolus. In *ATC*, 2012.

[38] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Ding Xuhua, and Robert Deng. ROPecker: A Generic and Practical Approach For Defending Against ROP Attacks. In *NDSS*, 2014.

[39] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A Detection Tool to Defend Against Return-oriented Programming Attacks. In *Proc. ACM Symposium on Information, Computer and Communications Security*, 2011.

[40] Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Commun. ACM*, May 1976.

[41] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. Quire: Lightweight Provenance for Smart Phone Operating Systems. In *USENIX Security Symposium*, 2011.

[42] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *SOSP*, 2005.

[43] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, 2011.

[44] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *USENIX Security Symposium*. USENIX Association, 2016.

[45] Sean Gallagher. How one rent-a-botnet army of cameras, DVRs caused Internet chaos. Ars Technica, Oct. 2016.

[46] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *NDSS*, 2004.

[47] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. ECDSA Key Extraction from Mobile Devices via Nonintrusive Physical Side Channels. In *Proc. ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[48] Daniel B. Giffin, Amit Levy, Deian Stefan, Alejandro Russo, David Terei, David Mazières, and John C. Mitchell. Hails: Protecting data privacy in untrusted web applications. In *OSDI*, 2012.

[49] Ian Goldberg, David Wagner, Randi Thomas, and Eric Brewer. A secure environment for untrusted helper applications (confining the wily hacker). In *USENIX Security Symposium*, 1996.

[50] Dan Goodin. 9 baby monitors wide open to hacks that expose users' most private moments. Ars Technica, Sep. 2015.

[51] Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proc. ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2012.

[52] Andy Greenberg. Marketing Firm Exactis Leaked a Personal Info Database With 340 Million Records. Retrieved Aug 2 2019, from `https://www.wired.com/story/exactis-database-leak-340-million-records/`.

[53] Andy Greenberg. Hackers Remotely Kill a Jeep on the Highway – With Me in It. Wired, Jul. 2015.

[54] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, Istvan Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *Proc. USENIX Security Symposium*, 2017.

[55] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *ACM Operating Systems Review*, 22(4), 1988.

[56] Weijia He, Maximilian Golla, Roshni Padhi, Jordan Ofek, Markus Dürmuth, Earlence Fernandes, and Blase Ur. Rethinking access control and authentication for the home internet of things (iot). In *Proc. USENIX Security Symposium*, 2018.

[57] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Janus: Intra-Process Isolation for High-Throughput Data Plane Libraries. Technical Report UR CSD/1004, 2018.

[58] Yu-Yang Hong, Yu-Ping Wang, and Jie Yin. NativeProtector: Protecting android applications by isolating and intercepting third-party native libraries. In *ICT Systems Security and Privacy Protection*, 2016.

[59] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *CCS*, 2016.

[60] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing Unique Code Target Property for Control-Flow Integrity. In *Proc. CCS*, 2018.

[61] Jie Huang, Oliver Schranz, Sven Bugiel, and Michael Backes. The ART of App Compartmentalization: Compiler-based Library Privilege Separation on Stock Android. In *CCS*, 2017.

[62] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *OSDI*, 2016.

[63] IBM Corporation. IBM Cloud. Retrieved Aug 2 2019, from https://www.ibm.com/cloud.

[64] Intel Corporation. Control-flow Enforcement Technology Specification. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.

[65] Intel Corporation. Intel IoT platform reference architecture white paper.

[66] Intel Corporation. Intel Architecture Memory Encryption Technologies Specification. https://software.intel.com/sites/default/files/managed/a5/16/Multi-Key-Total-Memory-Encryption-Spec.pdf. Accessed 17 Jun 2019.

[67] Intel Corporation. Intel Memory Protection Extensions (Intel MPX). https://software.intel.com/en-us/isa-extensions/intel-mpx. Accessed 29 Mar 2019.

[68] Intel Corporation. Intel Software Guard Extensions (Intel SGX). https://software.intel.com/en-us/sgx. Accessed 28 Mar 2019.

[69] Intel Corporation. Intel Software Guard Extensions (Intel SGX) SDK. https://software.intel.com/sgx-sdk. Accessed 28 Mar 2019.

[70] Intel Corporation. Intel 64 and IA-32 Architectures Software Developers Manual. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf, 2018. Accessed 29 Nov 2018.

[71] K. Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *NDSS*, 2000.

[72] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Z. Morley Mao, and Atul Prakash. Contexlot: Towards providing contextual integrity to appified iot platforms. In *NDSS*. Internet Society, 2017.

[73] H. Kawabata, T. Isohara, K. Takemori, A. Kubota, J. Kani, H. Agematsu, and M. Nishigaki. Sanadbox: Sandboxing third party advertising libraries in a mobile application. In *International Conference on Communications (ICC)*, 2013.

[74] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[75] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *Proc. EuroSys*, 2017.

[76] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. In *SOSP*, 2007.

[77] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proc. European Conference on Computer Systems*, 2017.

[78] Selena Larson. Verizon data of 6 million users leaked online. Retrieved Aug 2 2019, from https://money.cnn.com/2017/07/12/technology/verizon-data-leaked-online/index.html.

[79] Sanghak Lee, Jiwon Choi, Jihun Kim, Beumjin Cho, Sangho Lee, Hanjun Kim, and Jong Kim. FACT: Functionality-centric Access Control System for IoT Programming Frameworks. In *Proc. ACM Symposium on Access Control Models and Technologies*, 2017.

[80] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. Glamdring: Automatic Application Partitioning for Intel SGX. In *USENIX ATC*, 2017.

[81] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and

Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[82] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *OSDI*, 2016.

[83] Mark Russinovich. Introducing Azure confidential computing. Retrieved Aug 2 2019 from https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/.

[84] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proc. Hardware and Architectural Support for Security and Privacy*, 2013.

[85] Larry McVoy and Carl Staelin. Lmbench: Portable Tools for Performance Analysis. In *Proc. USENIX Annual Technical Conference*, 1996.

[86] Marcela S. Melara, Michael J. Freedman, and Mic Bowman. EnclaveDom: Privilege Separation for Large-TCB Applications in Trusted Execution Environments. http://arxiv.org/1907.13245, 2019.

[87] Marcela S. Melara, David H. Liu, and Michael J. Freedman. Protecting the IoT Against Data Leaks through Intra-Process Access Control (lightning talk). https://masomel.github.io/static/pubs/s&pDay-blitz-presentation.pdf, Oct 2017.

[88] Marcela S. Melara, David H. Liu, and Michael J. Freedman. Protecting the IoT Against Data Leaks through Intra-Process Access Control (poster). https://masomel.github.io/static/pubs/s&pDay-poster.pdf, Oct 2017.

[89] Marcela S. Melara, David H. Liu, and Michael J. Freedman. Pyronia: Redesigning Least Privilege and Isolation for the Age of IoT. http://arxiv.org/1903.01950, 2019.

[90] Microsoft Corporation. Microsoft Azure Cloud Computing Platform & Services. Retrieved Aug 2 2019 from http://docs.aws.amazon.com/iot/latest/developerguide/iot-security-identity.html.

[91] Microsoft Windows Dev Center. Mandatory Integrity Control. Accessed Feb. 11, 2019, https://docs.microsoft.com/en-us/windows/desktop/SecAuthZ/mandatory-integrity-control.

[92] A. C. Myers. JFlow: Practical mostly-static information flow control. In *ACM Symposium on Principles of Programming Language (POPL)*, 1999.

[93] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4), 2000.

[94] Nelly Porter, Jason Garms, Sergey Simakov. Introducing Asylo: an open-source framework for confidential computing. Retrieved Aug 2 2019 from https://cloud.google.com/blog/products/gcp/introducing-asylo-an-open-source-framework-for-confidential-computing.

[95] Nest Labs. Keeping data safe at Nest.

[96] Nvidia Corporation. Jetson Nano. Retrieved Jul. 24, 2019, from https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/.

[97] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In *USENIX Annual Technical Conference*, 2018.

[98] Capital One. Information on the Capital One Cyber Incident. Retrieved Aug 2 2019, from https://www.capitalone.com/facts2019/.

[99] OpenSSL Software Foundation. OpenSSL vulnerabilities. https://www.openssl.org/news/vulnerabilities.html. Accessed 19 Nov 2018.

[100] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software Abstraction for Intel Memory Protection Keys. http://arxiv.org/1811.07276, 2018.

[101] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. AdDroid: Privilege separation for applications and advertisers in android. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2012.

[102] Pete Loscocco. Security-enhanced Linux available at NSA site. Mailing list ARChive, accessed Aug 1, 2019, https://marc.info/?l=linux-kernel&m=97749381725894.

[103] Niels Provos. Improving host security with system call policies. In *USENIX Security Symposium*, 2003.

[104] Python Software Foundation. Sandboxed Python. Retrieved Feb. 7, 2018, from https://wiki.python.org/moin/SandboxedPython.

[105] A. Rahmati, E. Fernandes, K. Eykholt, and A. Prakash. Tyche: A Risk-Based Permission Model for Smart Homes. In *IEEE Cybersecurity Development (SecDev)*, 2018.

[106] Raspberry Pi Foundation. Raspberry Pi 4 Tech Specs. Retrieved Jul. 24, 2019, from https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/.

[107] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1), March 2012.

[108] Franziska Roesner and Tadayoshi Kohno. Securing embedded user interfaces: Android and beyond. In *USENIX Security Symposium*, 2013.

[109] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9), 1975.

[110] SAMSUNG. Device Protection and Trusted Code Execution.

[111] Roei Schuster, Vitaly Shmatikov, and Eran Tromer. Situational access control in the internet of things. In *Proc. ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[112] SELinux maintainers. SELinux Project wiki. Accessed Feb. 11, 2019, https://selinuxproject.org/page/Main_Page.

[113] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Taesoo Kim, Insik Shin, and X. Jiang. FlexDroid: Enforcing In-App Privilege Separation in Android. In *NDSS*, Feb. 2016.

[114] Jaebaek Seo, Byoungyoung Lee, Seongmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. Sgx-shield: Enabling address space layout randomization for sgx programs. In *NDSS*, 2017.

[115] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. AdSplit: Separating smartphone advertising from applications. In *USENIX Security Symposium*. USENIX Association, 2012.

[116] Youren Shen, Yu Chen, Kang Chen, Hongliang Tian, and Shoumeng Yan. To isolate, or to share?: That is a question for intel sgx. In *APSys*, 2018.

[117] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *Proc. Network and Distributed System Security Symposium*, 2017.

[118] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-tcb linux applications with SGX enclaves. In *Proc. Network and Distributed System Security Symposium*, 2017.

[119] S. Smalley and R. Craig. Security enhanced (SE) android: Bringing flexible MAC to android. In *NDSS*, 2013.

[120] Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. Protecting users by confining javascript with cowl. In *OSDI*, 2014.

[121] Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. Investigating user privacy in android ad libraries. 02 2019.

[122] Mengtao Sun and Gang Tan. NativeGuard: Protecting android applications from third-party native libraries. In *Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2014.

[123] Symantec Security Response. IoT devices being increasingly used for DDoS attacks. Symantec Official Blog, Sep. 2016.

[124] The MITRE Corporation. Common Vulnerabilities and Exposures (CVE) List. Retrieved Sep. 25, 2017, from https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=python.

[125] The PyPy Project. PyPy's sandboxing features. Retrieved Feb. 7, 2018, from http://pypy.readthedocs.io/en/latest/sandbox.html.

[126] Hongliang Tian, Yong Zhang, Chunxiao Xing, and Shoumeng Yan. SGXKernel: A Library Operating System Optimized for Intel SGX. In *Proc. Computing Frontiers Conference*, 2017.

[127] Yuan Tian, Nan Zhang, Yueh-Hsun Lin, XiaoFeng Wang, Blase Ur, XianZheng Guo, and Patrick Tague. SmartAuth: User-centered Authorization for the Internet of Things. In *Proc. USENIX Security Symposium*, 2017.

[128] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, and Peter Druschel. ERIM: Secure and Efficient In-process Isolation with Memory Protection Keys. http://arxiv.org/abs/1801.06822, 2018.

[129] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, Andre Dehon, and Jonathan Smith. BreakApp: Automated, Flexible Application Compartmentalization. In *NDSS*, 2018.

[130] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-based Fault Isolation. In *Proc. ACM Symposium on Operating Systems Principles*, 1993.

[131] Dan S. Wallach and Edward W. Felten. Understanding Java Stack Inspection. In *IEEE S&P*, 1998.

[132] Jun Wang, Xi Xiong, and Peng Liu. Between Mutual Trust and Mutual Distrust: Practical Fine-grained Privilege Separation in Multithreaded Applications. In *USENIX ATC)*, 2015.

[133] Yifei Wang, Srinivas Hariharan, Chenxi Zhao, Jiaming Liu, and Wenliang Du. Compac: Enforce component-level access control in android. In *Conference on Data and Application Security and Privacy (CODASPY)*, 2014.

[134] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proc. CCS*, 2012.

[135] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian Memory Protection. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[136] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: Memory Isolation for Linux Using Mondriaan Memory Protection. In *Proc. ACM Symposium on Operating Systems Principles*, 2005.

[137] Yongzheng Wu, Sai Sathyanarayan, Roland H.C. Yap, and Zhenkai Liang. Codejail: Application-transparent Isolation of Libraries with Tight Program Interactions. In *ESORICS*, 2012.

[138] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical policy enforcement for android applications. In *USENIX Security Symposium*, 2012.

[139] Yuanzhong Xu, Alan M. Dunn, Owen S. Hofmann, Michael Z. Lee, Syed Akbar Mehdi, and Emmett Witchel. Application-Defined Decentralized Access Control. In *USENIX ATC*, 2014.

[140] Alexander Yip, Xi Wang andNickolai Zeldovich, and M. Frans Kaashoek. Improving application security with data flow assertions. In *SOSP*, 2009.

[141] Aydan R. Yumerefendi, Benjamin Mickle, and Landon P. Cox. Tightlip: Keeping applications from spilling the beans. In *NSDI*, 2007.

[142] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *OSDI*, 2006.

[143] Kim Zetter. It's Insanely Easy to Hack Hospital Equipment. Wired, Apr. 2014.

[144] Xiao Zhang, Amit Ahlawat, and Wenliang Du. AFrame: Isolating advertisements from mobile applications in android. In *29th Annual Computer Security Applications Conference (ACSAC)*, 2013.

[145] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. ARMlock: Hardware-based Fault Isolation for ARM. In *Proc. ACM Conference on Computer and Communications Security*, 2014.

[146] Zope Foundation. Welcome to RestrictedPython's documentation! Retrieved Feb. 7, 2018, from http://restrictedpython.readthedocs.io/en/latest/.

Appendix A

CVE Reports for Python Libraries

Our analysis of reported Python library vulnerabilities in §3.2.1 covers Common Vulnerabilities and Exposures (CVE) reports made between January 2012 and March 2019. We found 123 reports for a total of 78 different Python libraries and frameworks in this seven-year time frame, and we identified nine main attack classes. Table A.1 shows the attack class and affected Python library or framework for each analyzed CVE report.

Table A.1: Reported Python library vulnerabilities between Feb 2012 and June 2019.

CVE Report	**Vulnerability Class**	**Affected Library/Framework**
CVE-2019-9948	Direct data leak	urllib*
CVE-2019-9947	Web attack	urllib*
CVE-2019-9740	Web attack	urllib*
CVE-2019-7537	Arbitrary code execution	donfig
CVE-2019-6690	Direct data leak	python-gnupg
CVE-2019-5729	MITM	splunk-sdk
CVE-2019-3575	Arbitrary code execution	sqla-yaml-fixtures
CVE-2019-3558	DoS	Facebook Thrift
CVE-2019-2435	Direct data leak	Oracle MySQL Connectors
CVE-2019-13611	Web attack	python-engineio
CVE-2019-12761	Arbitrary code execution	pyXDG
CVE-2019-11324	MITM	urllib*
CVE-2019-11236	Web attack	urllib*
CVE-2018-5773	Web attack	python-markdown2
CVE-2018-20325	Arbitrary code execution	definitions
CVE-2018-18074	Direct data leak	requests
CVE-2018-17175	Direct data leak	marshmallow
CVE-2018-10903	Direct data leak	python-cryptography
CVE-2018-1000808	DoS	pyopenssl
CVE-2018-1000807	DoS	pyopenssl
CVE-2017-9807	Arbitrary code execution	OpenWebif
CVE-2017-7235	Arbitrary code execution	cloudflare-scrape
CVE-2017-3590	Authentication bypass	MySQL

Table A.1 Continued.

CVE Report	**Vulnerability Class**	**Affected Library/Framework**
CVE-2017-2809	Arbitrary code execution	ansible-vault
CVE-2017-2809	Arbitrary code execution	tablib
CVE-2017-2592	Direct data leak	python-oslo-middleware
CVE-2017-16764	Arbitrary code execution	Django
CVE-2017-16763	Arbitrary code execution	Confire
CVE-2017-16618	Arbitrary code execution	OwlMixin
CVE-2017-16616	Arbitrary code execution	PyAnyAPI
CVE-2017-16615	Arbitrary code execution	MLAlchemy
CVE-2017-1002150	Web attack	python-fedora
CVE-2017-1000433	Authentication bypass	pysaml
CVE-2017-1000246	Weak crypto	pysaml
CVE-2017-0906	Web attack	recurly
CVE-2016-9910	Web attack	html5lib
CVE-2016-9909	Web attack	html5lib
CVE-2016-9015	MITM	urllib*
CVE-2016-7036	Weak crypto	python-jose
CVE-2016-5851	Web attack	python-docx
CVE-2016-5699	Web attack	urllib*
CVE-2016-5598	Direct data leak	MySQL
CVE-2016-4972	Arbitrary code execution	python-muranoclient
CVE-2016-2533	DoS	PIL
CVE-2016-2166	Weak crypto	Apache QPid Proton
CVE-2016-1494	Weak crypto	python-rsa
CVE-2016-0772	Weak crypto	smtplib
CVE-2015-7546	Authentication bypass	python-keystoneclient
CVE-2015-5306	Arbitrary code execution	ironic-inspector
CVE-2015-5242	Arbitrary code execution	swiftonfile
CVE-2015-5159	DoS	python-kdcproxy
CVE-2015-3220	DoS	tlslite
CVE-2015-3206	DoS	python-kerberos
CVE-2015-2674	MITM	restkit
CVE-2015-2316	DoS	Django
CVE-2015-1852	MITM	python-keystoneclient
CVE-2015-1326	Arbitrary code execution	python-dbusmock
CVE-2014-9365	MITM	urllib*
CVE-2014-8165	Arbitrary code execution	powerpc-utils-python
CVE-2014-7144	MITM	python-keystoneclient
CVE-2014-4616	Direct data leak	simplejson
CVE-2014-3995	Web attack	Django
CVE-2014-3994	Web attack	Django
CVE-2014-3598	DoS	PIL
CVE-2014-3589	DoS	PIL

Table A.1 Continued.

CVE Report	**Vulnerability Class**	**Affected Library/Framework**
CVE-2014-3539	Arbitrary code execution	rope
CVE-2014-3146	Web attack	lxml
CVE-2014-3137	Authentication bypass	Bottle
CVE-2014-3007	Arbitrary code execution	PIL
CVE-2014-1934	Symlink attack	eyeD3
CVE-2014-1933	Symlink attack	PIL
CVE-2014-1932	Symlink attack	PIL
CVE-2014-1929	Arbitrary code execution	python-gnupg
CVE-2014-1928	Arbitrary code execution	python-gnupg
CVE-2014-1927	Arbitrary code execution	python-gnupg
CVE-2014-1839	Symlink attack	logilab-common
CVE-2014-1838	Symlink attack	logilab-common
CVE-2014-1830	Direct data leak	requests
CVE-2014-1829	Direct data leak	requests
CVE-2014-1624	Symlink attack	python-xdg
CVE-2014-1604	Data spoofing	python-rply
CVE-2014-0472	Arbitrary code execution	Django
CVE-2014-0105	Authentication bypass	python-keystoneclient
CVE-2013-7459	Arbitrary code execution	PyCrypto
CVE-2013-7440	MITM	ssl
CVE-2013-7323	Arbitrary code execution	python-gnupg
CVE-2013-6491	MITM	python-qpid
CVE-2013-6444	MITM	pyWBEM
CVE-2013-6418	MITM	pyWBEM
CVE-2013-6396	MITM	python-swiftclient
CVE-2013-4482	Authentication bypass	python-paste-script
CVE-2013-4347	Weak crypto	python-oauth2
CVE-2013-4346	Auth token replay attack	python-oauth2
CVE-2013-4238	MITM	ssl
CVE-2013-4111	MITM	python-glanceclient
CVE-2013-2191	MITM	python-bugzilla
CVE-2013-2132	DoS	pymongo
CVE-2013-2131	DoS	python-rrdtool
CVE-2013-2104	Auth token replay attack	python-keystoneclient
CVE-2013-2013	Direct data leak	python-keystoneclient
CVE-2013-1909	MITM	Apache QPid Proton
CVE-2013-1665	Web attack	xml
CVE-2013-1664	DoS	xml
CVE-2013-1445	Weak crypto	PyCrypto
CVE-2013-1068	Authentication bypass	python-nova, python-cinder
CVE-2012-5825	MITM	Tweepy
CVE-2012-5822	MITM	zamboni

Table A.1 Continued.

CVE Report	**Vulnerability Class**	**Affected Library/Framework**
CVE-2012-5563	Authentication bypass	python-keystoneclient
CVE-2012-4571	Weak crypto	python-keyring
CVE-2012-4520	Web attack	Django
CVE-2012-4406	Arbitrary code execution	python-swiftclient
CVE-2012-3533	MITM	ovirt-engine-python-sdk
CVE-2012-3458	Weak crypto	Beaker
CVE-2012-3444	DoS	Django
CVE-2012-3443	DoS	Django
CVE-2012-2921	DoS	python-feedparser
CVE-2012-2417	Weak crypto	PyCrypto
CVE-2012-2374	Web attack	tornado
CVE-2012-2146	Weak crypto	elixir
CVE-2012-1575	Web attack	Cumin
CVE-2012-1502	Arbitrary code execution	PyPAM
CVE-2012-1176	DoS	PyFriBidi
CVE-2012-0878	Authentication bypass	python-paste-script

Appendix B

Open-Source Python IoT Applications

Table B.1 describes the sources of the 85 IoT applications we analyzed in §3.2, as well as the `plant_watering` application we studied in §3.6.

Table B.1: Sources for the open-source Python IoT applications analyzed in §3.2.2.

App Name	IoT Category	Source
babynap.py	audio	`www.hackster.io/memeka/baby-nap-night-activity-program-aa5ab0`
AlexaPiMQTTSmartHub	audio	`www.hackster.io/fileark/alexapi-mqtt-smart-hub-984dcf`
DIYAmazonAlexa	audio	`www.hackster.io/matrix-creator-team/build-a-diy-amazon-s-alexa-with-a-raspberry-pi-and-a-matr-8ad22d`
BlokBot.py	audio	`www.thingiverse.com/thing:1706105`
AlexaWithRaspioProHat	audio	`www.instructables.com/id/Build-an-Alexa-With-Raspio-Pro-Hat-and-Raspberry-P/`
RPiPersonalAssistant	audio	`www.instructables.com/id/Raspberri-Personal-Assistant/`
PiTranslate	audio	`www.daveconroy.com/turn-raspberry-pi-translator-speech-recognition-playback-60-languages/`

Table B.1 Continued.

App Name	IoT Category	Source
VoiceControlRoombaHue	audio	makezine.com/projects/use-raspberry-pi-for-voice-control/
GettingStartedWithReSpeaker	audio	www.cnx-software.com/2016/08/27/getting-started-with-respeaker-wifi-iot-board-audio-capabilities-voice-recognition-and-synthesis/
ControlTVWithVoice	audio	kazuar.github.io/raspberry_pi_voice_recognition/
JarvisInPython.py	audio	pythonspot.com/en/personal-assistant-jarvis-in-python/
ReSpeakerMessenger	audio	www.hackster.io/krvarma/respeaker-messenger-27e5ca
HeyAthena	audio	hackaday.io/project/9413-hey-athena-jarviss-better-half
PiHomeMaster	audio	www.hackster.io/25439/voice-controlled-smart-home-36e37e
AlexaPhone	audio	www.instructables.com/id/1970s-Raspberry-Pi-Amazon-AlexaPhone/
Alexabot	audio	www.instructables.com/id/Alexabot-Amazon-Alexa-Controlled-Robot-With-the-Ra/
TalkingGanesha	audio	www.instructables.com/id/Raspberry-Pi-based-answering-Ganesha/
soundSensorPhilippsHue.py	audio	www.instructables.com/id/Using-a-sound-sensor-with-a-Raspberry-Pi-to-contro/
STTIntelIoT.py	audio	www.instructables.com/id/Speech-to-Text/

Table B.1 Continued.

App Name	IoT Category	Source
AudioServer&Recorder.py	audio	www.instructables.com/id/Audio-Server-and-Recorder-With-Intel-Edison/
RPiAlexa	audio	www.instructables.com/id/Rasberry-Pi-Alexa/
AWShome	audio	www.hackster.io/awshome/awshome-home-automation-using-rpi-alexa-iot-a3d3dc
AlexaCoffeeMachine	audio	www.hackster.io/bastiaan-slee/coffee-machine-amazon-alexa-raspberry-pi-cbc613
VintageIntercom	audio	www.hackster.io/nick-brewer/vintage-intercom-echo-5d90bd
PiIndoorAirQualityMonitor	env	hackaday.io/project/2636-raspberry-pi-liv-pi-indoor-air-quality-monitor
SensorianIoTDashboard.py	env	hackaday.io/project/6018-sensorian-iot-dashboard
IoTRPiArduino	env	www.hackster.io/mjrovai/iot-connecting-the-rpi-arduino-and-the-world-165c54
NetworkMonitoringWithAWSIoT.py	env	www.hackster.io/phantom-formula-e97912/network-monitoring-with-aws-iot-b8b57c
CyberSafe	env	www.hackster.io/Satyavrat/cybersafe-your-personal-cloud-iot-platform-7caf02
EchoPiWeather	env	www.hackster.io/xelfer/nick-s-house-echo-pi-weather-b08dde

Table B.1 Continued.

App Name	IoT Category	Source
detect_bluetooth.py	env	www.hackster.io/cw-earley/simple-bluetooth-device-detection-0d2469
intelligentDoor.py	env	www.hackster.io/vijayenthiran/intelligent-door-e59113
RPiPIRSensor.py	env	www.hackster.io/engininer14/raspberry-pi-controlled-pir-sensor-with-email-notifications-0a8588
HomeControlCenterBBGW	env	www.thingiverse.com/thing:1596821
SitSat	env	www.instructables.com/id/SitSat-My-Personal-Office-Chair-Sitting-Pattern-Mo/
beerfridge	env	www.instructables.com/id/Raspberry-Pi-Beer-Fridge-of-Awesomeness/
thermostatRaspBerryPi	env	www.instructables.com/id/Thermostat-Raspberry-Pi/
WirelessDoorbell.py	env	www.instructables.com/id/Wireless-Doorbell-Raspberry-PI-Amazon-Dash/
IoTTempHumidityMonitor	env	www.instructables.com/id/Raspberry-Pi-IoT-Temperature-and-Humidity-Monitor/
EmailiPhoneNetworkPresence.py	env	www.instructables.com/id/Send-EmailTXT-when-you-return-home-detects-when-iP/
DHT22Logger	env	www.instructables.com/id/Raspberry-PI-and-DHT22-temperature-and-humidity-lo/

Table B.1 Continued.

App Name	IoT Category	Source
homeRoomTempHumMonitor.py	env	www.instructables.com/id/Home-Room-Temprature-and-Humidity-Monitor-with-Web/
weatherStation.py	env	www.instructables.com/id/Raspberry-Pi-Weather-Station/
smartScale	env	www.instructables.com/id/A-Raspberry-Pi-Weight-Tracking-Wise-Cracking-IoT-B/
HomeVentilation	env	www.instructables.com/id/Home-Ventilation/
PiRoomTempMonitorGnuplot.py	env	www.instructables.com/id/Raspberry-Pi-controlled-room-temperature-monitorin/
grove_IOT.py	env	www.instructables.com/id/Home-Environment-Monitor/
HomeTempThingspeak.py	env	www.instructables.com/id/Home-Temperature-Monitoring-Using-Raspberry-Pi-and/
SecurityCam	visual	www.pubnub.com/blog/2015-06-30-create-realtime-raspberry-pi-security-camera-w-parse/
pi_home_surveillance	visual	www.pyimagesearch.com/2015/06/01/home-surveillance-and-motion-detection-with-the-raspberry-pi-python-and-opencv/
GPIO&PiCameraUsingTelegram.py	visual	www.hackster.io/idreams/control-gpio-and-pi-camera-using-raspberry-pi-telegram-app-3a776a

Table B.1 Continued.

App Name	IoT Category	Source
SendingReceivingPicsMqtt.py	visual	developer.ibm.com/recipes/tutorials/sending-and-receiving-pictures-from-a-raspberry-pi-via-mqtt/
HomeSecurityEmailAlert.py	visual	www.instructables.com/id/Home-Security-Email-Alert-System-using-Raspberry-P/
PiScream.py	visual	hackaday.io/project/5721-piscream
surocam.py	visual	hackaday.io/project/2483-surveillance-robot-camera-a
IoTFishtank	visual	hackaday.io/project/7173-internet-connected-fishtank
pidoorbell.py	visual	www.hackster.io/pidoorbell-team/pidoorbell-7ef917
monitordoorbell.py	visual	www.hackster.io/Hoefnix/monitoring-the-doorbell-6a2000
RPiPhotobooth	visual	www.hackster.io/kevino/raspberry-pi-photo-booth-for-your-next-party-b8e76d
smartJpegCampera.py	visual	www.instructables.com/id/Smart-JPEG-Camera-for-Home-Security/
RPiLaserSecurity.py	visual	www.instructables.com/id/Raspberry-Pi-Laser-Security-System/
CodeYourOwnPhotobooth.py	visual	www.instructables.com/id/Lininger-Rood-Photo-Booth/
RPiIoTDoorbell	visual	www.instructables.com/id/Raspberry-Pi-IoT-Doorbell/

Table B.1 Continued.

App Name	IoT Category	Source
WhoIsAtCoffeeMachine	visual	www.instructables.com/id/Who-Is-at-the-Coffee-Machine-Facial-Recognition-Us/
PiNoculars.py	visual	www.instructables.com/id/PiNoculars-Raspberry-Pi-Binoculars/
WifiPhotobooth	visual	www.instructables.com/id/Wifi-Photobooth-With-a-Raspberry-Pi/
EasyRPiSecCam.py	visual	www.instructables.com/id/Easy-Raspberry-Pi-Security-Cam-With-Automatic-Web-/
LegoRobot	visual	www.instructables.com/id/Dog-Bot-Lego-Robot-Rover-With-Webcam/
camera-vision-logo.py	visual	www.instructables.com/id/Use-the-Raspberry-Pi-Camera-to-Detect-Company-Logo/
TwitterPhotoLiveFeed.py	visual	www.instructables.com/id/Twitter-Photo-Live-Feed/
vintage_raspi_cam	visual	www.instructables.com/id/Vintage-Raspberry-Pi-Camera/
360_pi_cam	visual	www.instructables.com/id/Uber-Cheap-360-Video-Camera/
RefrigeratorSecurity	visual	www.instructables.com/id/Facial-Recognition-Security-System-for-a-Refrigera/
DIYSmartphone	multi	www.instructables.com/id/Build-Your-Own-Smartphone/
AndyBot	multi	hackaday.io/project/1205-andy-a-multi-purpose-humanoid-robot

Table B.1 Continued.

App Name	IoT Category	Source
BeagleAlexa	multi	www.hackster.io/fcooper27/beaglealexa-56f174
SmartMirror	multi	www.instructables.com/id/Raspberry-Pi-Smart-Mirror/
tweetPic.py	multi	hackaday.io/project/11688-iot-twitter-sentry-ward-using-intel-edison
tweetingPlants.py	multi	www.hackster.io/archieroques/iot-tweeting-plants-w-raspberry-pi-f99da2
GardeningService	multi	www.hackster.io/terren/simple-gardening-service-manage-indoor-gardens-using-iot-be95d1
RPiSecuritySystem	multi	www.hackster.io/FutureSharks/raspberry-pi-security-system-with-motion-detection-camera-bed172
MarsRover	multi	www.instructables.com/id/Mobile-Station-Prototype-for-Environmental-Data-Ca/
HAL9000	multi	www.instructables.com/id/RaspberryPI-HAL9000/
SamsungARKTIKPhotobooth.py	multi	www.artik.io/blog/2016/07/building-photo-booth-samsung-artik-10/
devicePresenceAlert	multi	www.hackster.io/imrehg/device-presence-alert-home-automation-148276

Appendix C

Pyronia Case Study Policies

Listings C.1, C.2, and C.3 show the function-level access rules for each case study application analyzed in §3.6. The AppArmor policy for application-level access control (Listing C.4) is used to generate the default access rules for all applications, as described in §3.6.2.

Listing C.1: `twitterPhoto` function-level access policy rules.

```
tweepy.api.update_with_media /home/pyronia/libpyronia/apps/twitterPhoto/tomato
    -status.jpg ,
tweepy.<module> network 0:0:0:0:0:0:0:1 ,
tweepy.api.update_with_media network 10.0.2.3 ,
tweepy.api.update_with_media network 127.0.0.*,
tweepy.api.update_with_media network 104.244.42.*,
d /home/pyronia/libpyronia/apps/twitterPhoto/,
d /home/pyronia/libpyronia/apps/twitterPhoto/twitterPhoto.py ,
d /home/pyronia/libpyronia/apps/twitterPhoto/api_keys.py ,
d /home/pyronia/libpyronia/apps/twitterPhoto/api_keys.pyc ,
```

Listing C.2: `alexa` function-level access policy rules.

```
memcache.Client.__init__ /tmp/memcached.sock ,
requests.api.get network 127.0.0.53 ,
requests.api.get network 52.94.*,
requests.api.get network 52.119.*,
requests.api.get network 54.239.*,
requests.api.get network 72.21.207.*,
requests.api.post network 127.0.0.53 ,
requests.api.post network 54.239.*,
requests.api.post network 52.94.*,
requests.api.post network 52.119.*,
requests.api.post network 72.21.207.*,
d /home/pyronia/libpyronia/apps/alexa/,
d /home/pyronia/libpyronia/apps/alexa/alexa.py ,
d /home/pyronia/libpyronia/apps/alexa/creds.py ,
d /home/pyronia/libpyronia/apps/alexa/creds.pyc ,
d /home/pyronia/libpyronia/apps/alexa/recording.wav ,
d /home/pyronia/libpyronia/apps/alexa/response.mp3 ,
```

Listing C.3: `plant_watering` function-level access policy rules.

```
paho.mqtt.client.__init__ network 127.0.0.1 ,
paho.mqtt.client.tls_set /home/pyronia/libpyronia/apps/plant_watering/
    deadbeeeef-certificate.pem.cert.txt ,
paho.mqtt.client.tls_set /home/pyronia/libpyronia/apps/plant_watering/
    deadbeeeef-private.pem.key ,
paho.mqtt.client.tls_set /home/pyronia/libpyronia/apps/plant_watering/
    deadbeeeef-public.pem.key ,
paho.mqtt.client.tls_set /home/pyronia/libpyronia/apps/plant_watering/
    AmazonRootCA1.pem,
paho.mqtt.client.connect network 127.0.0.53 ,
paho.mqtt.client.connect network 34.21*,
paho.mqtt.client.connect network unknown,
paho.mqtt.client.connect network 52.*,
paho.mqtt.client.connect network 0.0.0.0 ,
paho.mqtt.client.connect network 35.*,
d /home/pyronia/libpyronia/apps/plant_watering/,
d /home/pyronia/libpyronia/apps/plant_watering/plant_watering.py ,
d /usr/bin/x86_64-linux-gnu-gcc-7,
d /sbin/ldconfig*,
d /bin/dash,
d /bin/sh,
d /bin/uname,
```

Listing C.4: Supporting application-level AppArmor policy rules common to all applications.

```
/lib/x86_64-linux-gnu/libpyronia.so rm,
/lib/x86_64-linux-gnu/libpyronia_opt.so rm,
/lib/x86_64-linux-gnu/libsmv.so rm,
/lib/x86_64-linux-gnu/libpthread-2.27.so rm,
/lib/x86_64-linux-gnu/libnl-genl-3.so.200.24.0 rm,
/lib/x86_64-linux-gnu/libnl-3.so.200.24.0 rm,
/etc/ld.so.cache r,
/etc/mime.types rm,
/etc/nsswitch.conf rm,
/etc/host.conf rm,
/run/resolvconf/resolv.conf r,
/run/systemd/resolve/stub-resolv.conf r,
/etc/hosts r,
/etc/gai.conf rm,
/etc/ssl/openssl.cnf r,
/lib/x86_64-linux-gnu/** rm,
/usr/lib/x86_64-linux-gnu/** rm,
/proc/*/net/psched r,
/proc/*/mounts rix,
/proc/*/status r,
/proc/*/maps r,
/dev/ r,
/dev/pts/* r,
/dev/urandom r,
/dev/random r,
/dev/null rw,
/usr/lib/locale/locale-archive r,
/usr/local/lib/python2.7/site-packages/ r,
/usr/local/lib/python2.7/site-packages** rmix,
/usr/local/lib/python2.7/dist-packages/ r,
/usr/share/zoneinfo/posixrules r,
/sys/devices/system/cpu/ rm,
/tmp/* rw,
/home/ r,
/home/pyronia/libpyronia/ r,
/home/pyronia/libpyronia/src/libpyronia.so rm,
/home/pyronia/cpython/ r,
/home/pyronia/cpython/Lib** rwmix,
/home/pyronia/cpython/pyronia_build/** rm,
/home/pyronia/cpython/pyronia_build/python rix,
/home/pyronia/.local/lib/python2.7/site-packages** rmix,
/home/pyronia/cpython/profiles/home.pyronia.cpython.pyronia_build.python-
    lib.prof r,
network inet tcp,
network inet dgram,
network inet6 stream,
network inet6 dgram,
```