The dissertation of Shen Liu was reviewed and approved by the following:

Gang Tan
Associate Professor of Computer Science and Engineering
Dissertation Advisor, Chair of Committee

Trent R. Jaeger
Professor of Computer Science and Engineering

Danfeng Zhang
Assistant Professor of Computer Science and Engineering

Peng Liu
Professor of Information Science and Technology

Chita R. Das
Professor of Computer Science and Engineering
Department Head of Computer Science and Engineering
Abstract

Partitioning a security-sensitive application into least-privileged components and putting each into a separate protection domain have long been a goal of security practitioners and researchers. However, existing techniques suffer from several obstacles that prevent program partitioning from being adopted in practice. For example, in C/C++ programs, the presence of pointers makes calculating data dependence, a key step in program partitioning, difficult and hard to scale; furthermore, C-style pointers do not carry bounds information, making it impossible to automatically marshall and unmarshall pointer data when they are sent across the boundary of partitions. More importantly, traditional partitioning approaches, which are based on static analysis, cannot find the optimal boundary for partitioning automatically. Programmers still have to do lots of manual work (e.g., declassification) to adjust the partitioning boundary to balance between performance and security. Furthermore, past partitioning systems only support a single label (i.e., sensitive vs. non-sensitive). It is useful to extend the program partitioning approach to support multiple labels, since complicated software usually handles sensitive data from multiple sources.

In this dissertation, we introduce our work for solving these problems in privilege separation. First, we propose a set of techniques for supporting general pointers in automatic program partitioning. Our system, called PttrSplit [5], constructs a Program Dependence Graph (PDG) [6] for tracking data and control dependencies in the input program and employs a parameter-tree approach for representing data of pointer types; this approach is modular and avoids global pointer analysis. Pttrsplit performs selective pointer bounds tracking to enable marshalling/unmarshalling of pointer data. Furthermore, we develop a toolchain called Program-mandering(PM) [7], which can automatically find the optimal boundary in secure program partitioning, according to user-provided constraints. The optimal boundary is selected by solving an integer programming model that simultaneously considers both security and performance. As a result, functions in the sensitive module but right on the optimal boundary are automatically declassified. We use experiments to show that Pttrsplit and PM work effectively on real-world security programs. Finally, we also extend our framework to support more general parti-
tioning – partitioning an application into multiple modules instead of only two to protect different kinds of sensitive data separately.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>viii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>x</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>xii</td>
</tr>
<tr>
<td>Chapter 1</td>
<td></td>
</tr>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Chapter 2</td>
<td>6</td>
</tr>
<tr>
<td>Background and Related Work</td>
<td></td>
</tr>
<tr>
<td>2.1 Privilege Separation</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Privilege Separation Enforcement</td>
<td>8</td>
</tr>
<tr>
<td>2.3 Security Goals of Privilege Separation</td>
<td>9</td>
</tr>
<tr>
<td>Chapter 3</td>
<td>12</td>
</tr>
<tr>
<td>PtrSplit: Supporting General Pointers in Automatic Program</td>
<td></td>
</tr>
<tr>
<td>Partitioning</td>
<td></td>
</tr>
<tr>
<td>3.1 Summary</td>
<td>12</td>
</tr>
<tr>
<td>3.2 System Overview</td>
<td>13</td>
</tr>
<tr>
<td>3.3 PDG and Partitioning</td>
<td>16</td>
</tr>
<tr>
<td>3.3.1 Regular Nodes and Edges in PDGs</td>
<td>16</td>
</tr>
<tr>
<td>3.3.2 Parameter Trees</td>
<td>17</td>
</tr>
<tr>
<td>3.3.3 LLVM PDG Construction</td>
<td>19</td>
</tr>
<tr>
<td>3.3.4 PDG-Based Program Partitioning</td>
<td>22</td>
</tr>
<tr>
<td>3.4 Selective Pointer Bounds Tracking</td>
<td>24</td>
</tr>
<tr>
<td>3.5 Type Based Marshalling and Unmarshalling</td>
<td>26</td>
</tr>
<tr>
<td>3.6 Evaluation</td>
<td>31</td>
</tr>
</tbody>
</table>
Chapter 4

Program-mandering: Quantitative Privilege Separation

4.1 Summary
4.2 A Motivating Example
4.3 System Overview
4.4 Graph-Based Partitioning
  4.4.1 Graph partitioning
  4.4.2 Partitioning with integer programming
4.5 Metrics
  4.5.1 Definition of metrics
  4.5.2 Alternative metrics
4.6 Implementation
4.7 Evaluation
  4.7.1 Evaluation with thttpd
  4.7.2 Evaluation with wget
  4.7.3 Evaluation with telnet
  4.7.4 Evaluation with nginx
  4.7.5 Evaluation with Linux shadow-utils
  4.7.6 Vulnerabilities mitigated by partitioning

Chapter 5

Multi-module Privilege Separation

5.1 Motivation
5.2 Multi-module Partitioning Algorithm
5.3 Discussion
5.4 Evaluation
  5.4.1 Evaluation with thttpd
  5.4.2 Evaluation for wget
  5.4.3 Evaluation with telnet

Chapter 6

Conclusion and Future Works

6.1 Summary of Our Current Work
6.2 Future Work

Appendix

Critical Background Techniques in PM

1  Program-dependence-graph partitioning
2  Encoding optimal partitioning as IP
3  Measuring information flow
List of Figures

3.1 The workflow of our automatic program-partitioning framework (gray components belong to PtrSplit). .......................... 14

3.2 A toy C program that encrypts a plaintext. ......................... 15

3.3 A PDG snippet for our running example. For clarity, the graph uses a single node for the entire for loop in encrypt; in contrast, PtrSplit’s PDG construction breaks a loop into LLVM IR instructions and has one node for each IR instruction. The graph also omits labels on data-dependence edges. .......................... 19

3.4 Syntax of types and a type map from type names to their type definitions. ............................................. 20

3.5 Type-based parameter-tree building. ............................... 20

3.6 Parameter tree for head of type Node *. .......................... 21

3.7 Type-based marshalling. In the algorithm, we assume a set of basic utility functions: enc_typ(t) for encoding a type into a list of bytes; enc_int(n) for encoding an integer; enc_ptr(p(b,e)) for encoding a pointer; size(t) for the size of values in type t; read_mem(b,t) for reading a value of type t from memory at address b. ................. 28

3.8 A two-node circular linked list. ................................... 29
3.9 Type-based unmarshalling. Function dec_typ(l) is for decoding a type in the first bytes of l; dec_int(l) for decoding an integer; dec_ptr(l) for decoding a pointer; alloc(n) for allocating a buffer of size n; write_mem(b, v) for writing v at address b in memory.

4.1 A motivating example.

4.2 Call graph and partitions for Fig. 4.1 example.

4.3 System flow of PM.

4.4 Call graph and partitions for wget.

4.5 PDG and partitions for telnet.

5.1 A motivating example for MMP: oblivious transfer.

5.2 Partitioning a program with two kinds of sensitive data (use PM twice).

5.3 PDG Annotation for the OT example.

5.4 The information flow capacity of sensitive data j to module i. In this example, information j can flow into module i from 3 other modules, and $C_{i,j}$ is the sum of the three capacity values.
List of Tables

2.1 Comparison of privilege separation mechanisms .......................... 9

3.1 Partitioning results of security-sensitive programs. (Abbreviations: "Total pointers": total pointer variables in LLVM-IR; "BR pointers": bounds-required pointer variables; "PBT": pointer bounds tracking; "SPBT": selective pointer bounds tracking.) .......................... 31

3.2 Partitioning results for SPECCPU 2006 benchmarks (use a random global as sensitive variable). .................................................. 34

3.3 The random partitioning results for SPECCPU 2006 benchmarks. 35

4.1 Benchmarks used during evaluation......................................... 53

4.2 Partitioning choices for thttpd ............................................. 54

4.3 Partitioning choices for wget .............................................. 55

4.4 Partitioning choices for telnet; “N/A” are for partitions that were not implemented. .......................................................... 58

4.5 Partitioning choices for nginx; “N/A” are for partitions that were not implemented. .......................................................... 60

4.6 Partitioning choices for Linux shadow-utils ............................ 61

4.7 Mitigated vulnerabilities by different partitions. ....................... 61
5.1 Three-module partitioning choices for thttpd: round 1 (RCode means replicated code. We list RCode specifically because we may have SCode $\geq 100\%$ when SCode includes too much RCode). . . . . . . 71

5.2 Three-module partitioning choices for thttpd: round 2. . . . . . . . 71

5.3 Three-module partitioning choices for wget. . . . . . . . . . . . . . 72

5.4 Three-module partitioning choices for telnet. . . . . . . . . . . . . . 73
Acknowledgments

First of all, I would like to express my gratitude to my advisor Professor Gang Tan for the continuous support of my Ph.D study and research and for his inspiration and motivation. He has been a great advisor. I would never have made my Ph.D at Penn State without his guidance and support.

Besides my advisors, I would like to thank the members of my dissertation committee, Prof. Trent Jaeger, Prof. Danfeng Zhang and Prof. Peng Liu for their time, reviewing my dissertation and comments.

I was extremely lucky and honored to meet and work with my great labmates Ben Niu, Dongrui Zeng, Robert Brotzman, Yongzhe Huang, Sun Hyoung Kim, Ashley Huhman and Michael Norris during my graduate studies. Thanks to their help and support. I have also benefited greatly from the collaboration with my mentor David Tarditi during my internship at Microsoft Research. Last but not the least, I am truly thankful to all my Penn State friends for their help and friendship, although I cannot list their names here considering the page limit.

This research is based upon work supported by US NSF grants CNS-1801534, CNS-1408826, CCF-1723571, as well as a gift from Intel. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any of the above organizations or any person connected with them.
Chapter 1  
Introduction

Following the principle of least privilege, privilege separation in software refers to separating a software application into multiple partitions, each with its own set of privileges. Partitions are isolated so that the compromise of one partition does not directly lead to the compromise of other partitions. Control and data flows among partitions are realized by Remote-Procedure Calls (RPCs); data for an RPC are marshalled and sent to the callee, which unmarshalls the data, performs its computation, and sends the result back to the caller.

Privilege separating programs in low-level, type-unsafe languages such as C/C++ is especially beneficial to security because these programs are prone to attacks (e.g., attacks enabled by memory vulnerabilities). For instance, OpenSSH was refactored by Provos [9] et al. to have unprivileged monitor processes for handling user connections and one privileged server process. Another example is the microkernel operating-system design, in which a minimum amount of code is kept in the kernel and most OS functionalities are pushed outside. Yet another example is Google’s Chromium browser, which isolates each tab into a sandboxed process [10].

These restructuring efforts have significantly improved the security of the relevant software; however, they are labor intensive and sometimes error-prone. Several systems [11,13,14] have been proposed to apply program analysis to separate C/C++ applications automatically into partitions, from a small number of user annotations about sensitive data. These systems demonstrate automatic program partitioning can be practical. However, these systems suffer from several limitations, which hinder their practical use.

First, one major limitation of existing privilege separation systems is that they lack good support for pointer data, which are prevalent in C/C++ applications.
C-style pointers do not carry bounds information; when a pointer needs to be sent across the partition boundary in an RPC call, marshalling does not know the size of the underlying buffer and consequently cannot marshal the buffer automatically. Some systems adopt heuristics when marshalling pointer data (e.g., a “char *” pointer is assumed to point to a null-terminated string); however, programmers are often required to write marshalling and unmarshalling code manually for pointer data, especially for pointers that point to dynamically allocated buffers. Some systems avoid the problem by restricting the partitioning algorithm to not create partitions that require pointer passing; this design, however, limits the flexibility of where partitions can be created. Furthermore, a program-partitioning algorithm needs to reason about dependence in a program to decide where to split. When the program has pointers, a global pointer analysis is typically required to understand aliasing and how data flow in memory. However, global pointer analysis is often complex and does not scale to large programs.

Second, while security is the motivating goal in performing privilege separation, the performance implications of the resulting program must also be carefully considered. No matter how protection domains are isolated (e.g., via the OS process isolation or via some hardware mechanism such as Intel’s SGX), there is invariably a performance cost when data and control cross protection-domain boundaries; as a result, refactoring a monolithic application into multiple modules in different protection domains comes with a performance cost, incurred by changing local data/code accesses into remote data/code accesses, which cross the partition boundary. Importantly, the performance cost depends on how the application is partitioned; that is, how boundaries of modules are drawn in the application and what code is duplicated. If considering only performance, one would just put all code into one protection domain, reverting back to the original monolithic application; however, security would not be improved. Similarly, considering only security could result in bad performance.

Third, most separation systems create only two partitions, which is not enough to achieve the fine-grained privilege separation. For example, in an ftp server, sensitive data often come from different sources (e.g., a database and a networking socket); simply treating different kinds of sensitive information as a whole is too coarse-grained to achieve the least-privilege principle. Simply using existing separation systems to process each kind of sensitive data separately may lead to an
unsatisfactory solution.

Considering these challenges, a problem that is worth exploration is: can we implement a program partitioning framework that comprehensively evaluates multiple factors to support more automatic and more flexible privilege separation?

Our statement is: *program partitioning can be enhanced to make privilege separation more automatic, more flexible, and accordingly more practical.*

In this dissertation, we propose a series of techniques to improve the automatic privilege separation. Our approach can help solve the aforementioned problems.

To address the problem of supporting pointer data, we propose **PtrSplit**. PtrSplit provides a modular way of constructing a Program Dependence Graph (PDG) for a program with pointers. Unlike previous work, PtrSplit uses *parameter trees* to connect a caller function with its callee function. By using parameter trees, PtrSplit can reason about the dependence in a program to decide where to partition without using complex global pointer analysis. Instead, only intraprocedural pointer analysis is needed in the whole process, which greatly increases PtrSplit’s practicality for large-scale programs.

For pointer marshalling/unmarshalling in RPC, PtrSplit instruments the program so that pointers carry bounds information. To reduce the significant performance overhead caused by the full pointer bounds tracking, PtrSplit computes a set of pointers that cross the partitioning boundary, and only instruments these pointers to get their bounds information during the runtime. We call this method selective pointer bounds tracking.

To address the problem of balancing security and performance in privilege separation, we propose **Program-mandering (PM)** [7] to supports quantitative privilege separation. PM defines a set of metrics for quantifying security and performance, and allows quantitative tradeoffs among these metrics. For example, PM uses the amount of sensitive information that flows from the sensitive to the insensitive domain as a metric to quantify "security", and uses the frequency of context switches between the two domains to quantify "performance". We choose security metrics to reason about how well computed partitions enforce information flow control to: (1) protect the program from low-integrity inputs or (2) prevent leakage of program secrets.

Once the metrics for quantification are prepared, PM casts privilege separation as an optimization problem. This is achieved by solving an integer-programming
model that optimizes for a user-chosen metric while satisfying the remaining security and performance constraints on other metrics. Specifically, given an application, PM reuses PtrSplit’s PDG that is annotated with measurements on the metrics, collected via program analysis. Next, we transform the annotated PDG as well as programmer-specified constraints on partitioning factors into an Integer Programming (IP) model. A generic IP solver is then used to find the optimal partitioning according to a programmer-specified criterion (e.g., minimizing performance overhead).

PM is the first system that combines quantitative information flow with privilege separation. This not only provides a security metric that aligns well with security goals common in applications, but also reduces users’ burden of performing manual declassification—the optimal partition computed by PM automatically gives where data should be declassified. We have implemented PM and evaluated it on a set of real world programs. Our experience shows that PM helps users make quantitative trade-offs among multiple factors. After observing initial partitions, users could use PM to improve the balance between security and performance by setting simple constraints, in an iterative process.

To solve the problem of isolating and protecting different kinds of sensitive data separately, we propose Multiple-module partitioning (MMP) that supports partitioning a program into multiple modules, which is more general and fine-grained than most existing work that can only create two modules (sensitive vs. insensitive) for isolation. Based on the PDG and metrics for security and performance we already have in PtrSplit and PM, we annotate and measure \( n \) \( (n \geq 2) \) kinds of sensitive data in a program at the same time. Instead of analyzing the metrics between a sensitive module and an insensitive module in Program-mandering, we comprehensively consider the total amounts of metrics (security, performance, interface complexity) from a global perspective. As far as we know, MMP is the first work that achieves global optimal partitioning in privilege separation when we have multiple sensitive data for isolation and protection.

The remainder of the dissertation is organized as follows. Chapter 2 introduces related work about privilege separation and its enforcement mechanisms. Chapter 3 presents PtrSplit, our framework that supports general pointers in C program partitioning. Chapter 4 introduces Program-mandering (PM) that can help programmers find the optimal boundary to balance between performance and security.
Chapter 5 introduces Multiple-modules partitioning (MMP), which supports partitioning a monolithic program into multiple modules to achieve fine-grained privilege separation. Finally, we conclude the dissertation and discuss future work in Chapter 6.
Chapter 2  
Background and Related Work

In this chapter, we provide an overview of related work and briefly introduce existing techniques to protect sensitive code and data. First, we introduce the concept of privilege separation; second, we discuss several mainstream privilege separation enforcement mechanisms; next, we discuss some existing difficulties and challenges for privilege separation; finally, we describe some metrics for security and performance measurement, which is crucial to quantitative privilege separation.

2.1 Privilege Separation

The concept of program partitioning can be traced back to the 1980s. In 1981, Mark Weiser proposed the idea called program slicing [1]. A program slice is an executable program that is separated from the original program by removing statements, such that the separated program replicates part of the behavior of the original program. Program slicing is widely used in many fields in computer science. For example, computer architecture engineers can apply program slicing to split a sequential program into pieces that can be executed in parallel [2,3]. From security perspective, programmers can apply program slicing to automatically partition a monolithic program, and deploy either a confidentiality or integrity policy to protect the sensitive part.

Several tools have been proposed to assist programmers in program partitioning. Privman [4] is a library for helping programmers manually partition their applications to control access to privileged system calls. Wedge [11] provides a dynamic profiling tool for partitioning assistance. It collects statistics about how a program uses memory to help programmers draw partition boundaries; however, program-
mers still need to perform manual code changes and partitioning. Trellis [12] infers access policies on code and data in multi-user applications from user annotations and enforces the policies through a modified OS.

Automatic program partitioning employs program analysis and separates a program into multiple partitions, with minimum user involvement. Privtrans [13] performs static analysis to automatically partition a C application into a privileged master process with sensitive information and an unprivileged slave process. SeCage [20] employs hybrid static/dynamic analysis to compute a set of functions that can access secrets and isolates the sensitive partition via hardware virtualization support. Jif/split [15,17] automatically partitions a Java source program based on security-label annotations and a description of trust relationships between protection domains.

ProgramCutter [14], Swift [16], and SOAAP [22] are partitioning systems that consider both security and performance, but none of them enables a user to make quantitative tradeoffs between security and performance since they lack metrics for quantifying security. In detail, ProgramCutter collects system calls that a function makes and uses that information to isolate a set of functions that access a sensitive resource. It does not consider the impact of information flow on partitioning, which is critical to preventing data leakage and protecting data integrity. Swift separates web application code into two components, one that runs on the web client and one that runs on the web server. Swift computes a partition that minimizes the number of boundary crossings (i.e., between the client and server). However, it does not quantify security. Further, Swift relies on the Jif programming language for writing the initial program, guaranteeing that any partition will satisfy information flow requirements. However, writing programs to satisfy information flow comprehensively often creates a significant manual burden (e.g., to define and place declassifiers to resolve information flow errors). SOAAP is an interactive tool that asks a user to provide source-level annotations to guide partitioning. SOAAP’s annotation requirement is heavyweight; for example, it requires annotations about the partition boundary and what global state can be accessed by each security domain. While SOAAP includes a performance simulator to help users decide whether a partition would meet a performance goal, it does not quantify security nor does it provide a framework for users to explore the quantitative tradeoff between security and performance.
2.2 Privilege Separation Enforcement

After an application has been partitioned, privilege separation requires an enforcement mechanism that isolates partitions. Table 2.1 provides a comparison of privilege enforcement mechanisms for applications. Traditional privilege separation enforcement mechanisms use different processes for deployment. For instance, Privtrans performs static analysis to automatically partition a C application into a privileged master process with sensitive information and an unprivileged slave process. OpenSSH was refactored by Provos et al. to have unprivileged monitor processes for handling user connections and one privileged server process [23]. Another example is the microkernel operating-system design, in which a minimum amount of code is kept in the kernel and most OS functionalities are pushed outside. Yet another example is Google’s Chromium browser, which isolates each tab into a sandboxed process [26].

Privilege separation can also be enforced by using virtual machines. The sensitive code can directly run as independent processes in separate virtual machines to achieve strict access control. SeCage [20] retrofits commodity hardware virtualization extensions to support isolation. Dune [27] uses Intel VT-x x86 virtualization ISA extensions to implement separation. TrustVisor [28] achieves sensitive code protection by using a tiny hypervisor.

With the emergence of Trusted Execution Environments (TEEs), there have also been program-partitioning frameworks that target Intel’s SGX or ARM’s TrustZone. For instance, Rubinov et al. [18] proposed a static-analysis framework that partitions an Android application into one component that runs in TrustZone’s secure world and one that runs in TrustZone’s normal world. A similar system called Glamdring [19] targets Intel’s SGX.

Intel provides Memory Protection Keys (MPK) for their recent CPUs. MPK introduces a register that contains a protection key for tagging memory. This new MPK technique can be utilized to implement privilege separation by tagging the sensitive memory regions. ERIM [29] implements efficient in-process isolation using MPK to protect both confidentiality and integrity of in-memory data.
### Table 2.1. Comparison of privilege separation mechanisms

<table>
<thead>
<tr>
<th></th>
<th>Isolation within the same process</th>
<th>Hardware Enforced</th>
<th>Context-switch overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS processes</td>
<td>No</td>
<td>No</td>
<td>High</td>
</tr>
<tr>
<td>Virtual machines</td>
<td>No</td>
<td>Maybe</td>
<td>Very high</td>
</tr>
<tr>
<td>SFI</td>
<td>Yes</td>
<td>No</td>
<td>Low</td>
</tr>
<tr>
<td>SGX/TrustZone</td>
<td>Yes</td>
<td>Yes</td>
<td>Very high</td>
</tr>
<tr>
<td>Intel MPK</td>
<td>Yes</td>
<td>Yes</td>
<td>Low</td>
</tr>
</tbody>
</table>

### 2.3 Security Goals of Privilege Separation

Privilege separation has traditionally been applied to reduce the privileges of individual domains to achieve least privilege. For example, privilege separation for OpenSSH by Provos et al. [23] refactored the program into one privileged server and many unprivileged monitors. Access to secret keys is removed from the monitors. The server retains access to the files storing secret keys, but no longer needs network access. However, least privilege may still permit attacks from an unprivileged domain. For example, the SELinux policy for the server allowed it to access files modifiable by untrusted monitors, which allowed unauthorized information flows (i.e., from low-integrity monitors to high-integrity servers) that may enable attacks. Researchers suggested modifications to the SELinux policy and changes to access control enforcement to limit the channels (i.e., particular system call invocations) through which the server could access untrusted resources [24], approximating Clark-Wilson integrity [25]. In particular, Clark-Wilson integrity enforces an information-flow requirement that all low-integrity data received by a high-integrity program may be received only if the program can upgrade (e.g., endorse or filter) that data. Thus, a privilege-separation method must enable users to configure partitions that only allow authorized information flows.

The information-flow security requirements must be fulfilled in order to deploy sensitive domains that process low-integrity data and high-secrecy data. Low-integrity domains are used to receive untrusted, external input; the security goal is to protect the program as much as possible from such untrusted inputs. The purpose of high-secrecy domains is to provide access to program secrets; so the
security goal is to prevent leakage of such secrets from the program, even if part of the program comes under attacker control. As the low-integrity domain cannot be entrusted with any secret data, we prohibit any partition that enables the low-integrity domain to access secret data from the high-integrity domain or the file system. In order to protect the use of high-secrecy data, the integrity of the high-secrecy domain must be protected. We should aim to minimize the amount of low-integrity data received by the high-integrity (and high-secrecy) domains. However, in this case, the partitioning is to protect the high-secrecy and high-integrity functions, not to protect the program at large from functions that receive untrusted inputs. Thus, from a Clark-Wilson perspective, it is best to minimize the amount of code performing security-critical functionality to reduce its attack surface.

In order to deploy sensitive domains that process low-integrity data and high-secrecy data, the information-flow security requirements must be fulfilled as follows:

**Low-integrity domains.** Low-integrity partitions are created to receive untrusted, external input; the security goal is to protect the program as much as possible from such untrusted inputs. For that, we leverage PM to create a sensitive, low-integrity domain to collect such untrusted inputs and an insensitive high-integrity domain to guard itself from those untrusted inputs. When some low-integrity information is sent to the high-integrity domain, that information must be declassified (i.e., endorsed) in the high-integrity domain.

To guide PM to generate such partitions, we target noninterference, which has two implications for partitioning. First, low-integrity domains should have minimal code size. We want to minimize the amount of code that can be directly influenced by low-integrity data. Second, the quantity of information conveyed from the low-integrity domain to the high-integrity domain should be minimized to reduce the amount of data to endorse or filter. We note that the data may convey between domains directly, via RPCs, and through indirect channels, such as the file system. The former is controlled by where we place partition boundaries. The latter is controlled by the least privilege permissions needed to execute the partitions correctly. Ideally, the partitioning creates RPCs that convey minimal information, and least privilege permissions that do not require the high-integrity domain to use any data written to the file system.

As the low-integrity domain cannot be entrusted with any secret data, we
prohibit any partition that enables the low-integrity domain to access secret data from the high-integrity domain or the file system.

**High-secrecy domains.** The purpose of high-secrecy domains is to provide access to program secrets; so the security goal is to prevent leakage of such secrets from the program, even if part of the program comes under attacker control. Thus, we leverage PM to create a sensitive, high-secrecy domain to access secrets and an insensitive low-secrecy domain that must not have access to secret information. The high-secrecy domain must declassify any data to be sent via RPC to the low-secrecy one.

To guide the use of PM to generate such partitions, we again target noninterference, which aims to ensure that any low-secrecy programs will produce the same (low) outputs regardless of the high-secrecy data processing. Thus, we aim to minimize the information flow from the high-secrecy to the low-secrecy domain to reduce the amount of data that must be declassified. The partitioning boundary defines where the sensitive partition must declassify data. In addition, if the sensitive partition outputs the secret data to external resources, such as the file system, that partition must also declassify that data. Ideally, secret data is not output to the file system.

In order to protect the use of high-secrecy data, the integrity of the high-secrecy domain must be protected. We should aim to minimize the amount of low-integrity data received by the high-integrity (and high-secrecy) domains. However, in this case, the partitioning is to protect the high-secrecy and high-integrity functions, not to protect the program at large from functions that receive untrusted inputs. Thus, from a Clark-Wilson perspective, it is best to minimize the amount of code performing security-critical functionality to reduce its attack surface.
In this chapter, we introduce our PtrSplit work. PtrSplit includes a series of techniques that enable the support of pointers in automatic program partitioning. The prototype of PtrSplit is implemented inside LLVM; our preliminary evaluation on security-sensitive benchmarks and compute-intensive benchmarks suggests the system is already practical for C applications with pointers and can produce executable partitions with a modest amount of performance overhead. The following sections describe PtrSplit’s system design, key techniques and experimental evaluation.

3.1 Summary

In this section, we summarize PtrSplit’s major techniques and contributions are as follows:

- Taking source code as input, PtrSplit constructs a static Program-Dependence Graph (PDG) for the program. A feature in PtrSplit’s PDG that distinguishes it from previous PDGs for imperative programs is a technique called parameter trees. It provides a modular way of constructing the PDG for a program with pointers; as a result, only an intraprocedural pointer analysis is needed, instead of a global pointer analysis. Our tree representation generalizes the object-tree approach in prior work [40], which discussed a tree representation for objects
in object-oriented languages and did not cover pointers at the language level; our system uses the tree representation for representing pointers in imperative languages and deals with circular data structures resulting from pointers.

- To marshall pointers, PtrSplit instruments the program so that pointers carry bounds information. PtrSplit makes the critical observation that program partitioning does not need full pointer tracking—it is sufficient to track the bounds of pointers that cross the partitioning boundary. Therefore, given an arbitrary partitioning of the program,PtrSplit computes a set of pointers that require bounds information and instruments the program to track the bounds of only those pointers. We call this selective pointer bounds tracking.

- PtrSplit generates code that performs marshalling and unmarshalling for data sent over an RPC call. This is automatic even for pointer data because all pointers that cross the partition boundary carry bounds information. We describe a type-based algorithm for performing deep copies of pointer data, which can cope with the situation of circular data structures and arbitrary aliasing, without user involvement.

3.2 System Overview

Figure 3.1 presents.PtrSplit’s workflow. It takes the source code of a single threaded C application as input; the code has been annotated by the programmer with information about sensitive and declassified data. Sensitive data can be either confidential data (e.g., keys) or data from an untrusted source (i.e., tainted data such as user input).

The source code is converted to an LLVM IR program by LLVM’s front end. PtrSplit then constructs the PDG for the IR program. A PDG-based algorithm then computes two raw partitions: one sensitive raw partition that can access sensitive data and one insensitive raw partition with the rest of the code. However, raw partitions cannot run directly because after partitioning some function calls become Remote-Procedure Calls (RPCs) and it is necessary to add RPC wrapper code for data marshalling and unmarshalling. In PtrSplit, each partition is loaded into a separate process, so RPC wrapper code must be added for inter-process communication.
Figure 3.1. The workflow of our automatic program-partitioning framework (gray components belong to PtrSplit).

Based on raw partitions, PtrSplit performs selective pointer bounds tracking, which tracks bounds information for pointers whose values can potentially cross the partitioning boundary. Bounds information for pointers is then used by a type-based method, which generates RPC wrappers that perform data marshalling and unmarshalling for inter-process RPC calls. In the end, PtrSplit generates one executable partition with all sensitive code, data, and RPC wrappers, and also an executable partition with insensitive code, data, and RPC wrappers.

We will illustrate the main points of PtrSplit by a toy example in Figure 3.2. The example takes a username and a text input from the user, greets the user by the `greeter` function, initializes a key, and encrypts the text by xor-ing it with the key. The global `key` is the sensitive data that needs protection; therefore, it is marked sensitive using a C attribute. Note the program has a format-string vulnerability at line 6 in `greeter`, which could allow an attacker to take over the program and learn the key.

Intuitively, a partitioning framework should put the `greeter` function into the insensitive partition since no sensitive data can flow to it. Other functions, including `initkey`, `encrypt`, and `main` should be in the sensitive partition since `key` may be accessed by them directly or indirectly. This partitioning would isolate the format-string error in `greeter` into the insensitive partition and prevent the attacker from learning the key. Similar to other partitioning frameworks, PtrSplit also supports declassification. If `ciphertext` is annotated as declassified data,
main can also stay in the insensitive partition even though it accesses ciphertext; in this way, vulnerabilities in main are isolated.
3.3 PDG and Partitioning

Program partitioning requires analyzing dependence in an input program carefully and adjusting the program to a distributed programming style. A key step in PtrSplit is to construct for the program a graph representation of dependencies, called the Program Dependence Graph (PDG [6]). Conceptually, a PDG represents a program’s data and control dependence in a single graph and can facilitate static analysis including program slicing and automatic parallelization. There are many systems that construct PDGs for programs in different languages and with different precision. A distinguishing feature of our PDG construction is its approach of parameter trees for representing composite data (e.g., pointers) that are passed during function calls and returns. We will start explaining nodes and edges that are common in a PDG representation in Sec. 3.3.1, and discuss the parameter-tree approach in Sec. 3.3.2. In this discussion, we will use examples in C for readability, even though PtrSplit constructs PDGs for LLVM IR programs; the IR-level PDG construction will be explained in Sec. 3.3.3. Finally, we present a standard PDG-based partitioning algorithm in Sec. 3.3.4.

3.3.1 Regular Nodes and Edges in PDGs

Every instruction in the program is represented as an instruction node in a PDG. For edges, there are data/control dependence edges and call edges.

In general, an instruction node $n_1$ is data dependent on instruction node $n_2$ if $n_1$ uses some data produced by $n_2$. Our PDGs have two kinds of data-dependence edges:

1. There is a def-use dependence if $n_1$ uses a variable $x$ that is defined in $n_2$; an edge from $n_2$ to $n_1$ with label $x$ is added.\footnote{The edge direction reflects the dataflow direction, instead of the direction of dependence; this makes algorithms on PDGs easier to state.}

2. There is a RAW (Read-After-Write) dependence if $n_1$ reads memory that was written by $n_2$ and an edge from $n_2$ to $n_1$ is added with label $id$, assuming $id$ points to the memory in question.

An example of def-use dependence is as follows. Variable $x$ is defined in “$x = 1$” and later used in the assignment to $y$. 
x = 1;
... // x not modified
y = x + x;

An example of RAW dependence is as follows. Memory location pointed to by p is written in instruction “*p = 1” and read in the assignment to y.

*p = 1;
... // memory pointed to by p not modified
y = *p;

For control dependence, an instruction node n1 is control dependent on n2 if, intuitively, there are two edges out of n2 and taking one edge results in the execution of n1, while taking the other edge results in the case of not executing n1. The formal definition of control dependence can be found in [6].

Call edges connect call sites with the entries of possible callee functions. For an indirect call (a call through a variable, e.g.), it may be connected with multiple possible callee functions. We adopt static, type-based matching [30] so that an indirect call via a function pointer can target any function whose type is compatible with the function pointer’s type. For this method to be valid, some preprocessing of source code is required [30] (e.g., to eliminate type casts that involve function-pointer types by adding function wrappers).

### 3.3.2 Parameter Trees

The motivation for parameter trees is to simplify the computation of inter-procedural data dependence and obtain a modular PDG-construction approach. To illustrate this, let us revisit the example in Fig. 3.2. Notice on line 30, there is a function call to `encrypt` and `text` is passed; inside `main` the `text` buffer is written by `scanf` and inside `encrypt` the passed buffer is read. Therefore, there is a RAW dependence between the `scanf` call instruction in `main` and the instruction in `encrypt` that reads the buffer.

This kind of dependence is inter-procedural. The proper calculation of such dependence would require a whole program analysis such as a global pointer analysis, which needs access to all code. It is complex, and often does not scale. Furthermore, the resulting PDG may suffer from edge blow ups: suppose the caller
has \( n \) instructions that can write to a buffer and all \( n \) writes can affect the result of \( m \) reads in the callee, then the number of dependence edges is \( O(n \times m) \).

To obtain a modular and scalable PDG-construction system, we introduce parameter trees. In this approach, for each parameter of a function, we build a formal parameter tree according to the parameter’s type. The parameter tree contains nodes that represent all the storage (memory) regions that the function can access through the parameter directly or indirectly.

We will present a formal algorithm for parameter-tree building in Sec. 3.3.3. An example is instead discussed in this subsection. The parameter tree for the plaintext parameter of encrypt in the running example can be found inside Fig. 3.3. It has a root node labeled “plaintext:char*” for representing the storage of the pointer, and a child node labeled “*plaintext:char” for the memory region that the pointer points to. The type in a parameter-tree node specifies the type of elements the corresponding memory region holds.

In addition to formal parameter trees, we also construct an actual parameter tree for each argument at a function call site, and connect nodes in an actual tree with corresponding nodes in a formal tree by data-dependence edges. Fig. 3.3 draws a PDG snippet for the running example that shows the interaction between main and encrypt. The call site in main has two arguments: text and strlen(text); each is built with an actual parameter tree. The encrypt function has two parameters, each with a formal parameter tree.

Parameter trees enable modular construction of PDGs. To build a PDG for a large program, we can first build a PDG for each function using an intra-procedural analysis, and then “glue” the functions’ PDGs together using parameter trees. A global analysis is avoided. Put it in another way, all data-dependence edges become local, either between two instruction nodes or between an instruction node and a parameter-tree data node. Inter-procedural data dependence is represented transitively via local data-dependence edges. Let us revisit the running example; recall that there is a RAW dependence between the scanf call in main and the instruction in encrypt that reads the plaintext buffer; this interprocedural dependence is broken into three edges in Fig. 3.3: one from the scanf node to parameter-tree node *text; one from *text to parameter-tree node *plaintext; one from *plaintext to the loop node that reads memory via plaintext.

By using parameter trees, if the caller has \( n \) instructions that can write to a
buffer and all $n$ writes can affect the result of $m$ reads in the callee, the number of edges becomes $O(n + m)$: we add $O(n)$ edges from the write instructions to the data nodes in the actual parameter trees, $O(m)$ edges from the data nodes in the formal parameter trees to the $m$ read instruction, and a constant number of edges between actual and formal parameter trees.

We note that return values and global data are also represented as parameter trees. For instance, the `key` and `ciphertext` global data in the running example are represented using trees similar to the one for `plaintext`, as shown in Fig. 3.3. Also, data-dependence edges between instructions that perform global data access and corresponding tree nodes for global data are added.

### 3.3.3 LLVM PDG Construction

We next outline PtrSplit’s algorithm for PDG construction in LLVM. PtrSplit’s parameter tree building is type based. We next formalize the process. Fig. 3.4 presents the syntax of a subset of LLVM IR types. A type $t$ can be an integer type...
TM = \{\textbf{Node} \rightarrow \text{struct} \{\text{value} : \text{int}; \text{next} : \textbf{Node}\star\}\}

\begin{align*}
t &:= \text{int} \mid t_1 \star \mid \text{struct} \{i_1 : t_1; \ldots; i_m : t_m\} \mid tn \\
\text{TM} &:\text{TypeName} \rightarrow_{fin} \text{Type}
\end{align*}

Figure 3.4. Syntax of types and a type map from type names to their type definitions.

\begin{align*}
\text{buildTree}(t, id, k) &= \begin{cases} \\
\text{Tree}(id : t) & \text{if } t = \text{int} \\
\text{Tree}(id : t, tr_1) & \text{if } t = t_1\star \\
& \quad \text{and } tr_1 = \text{buildTree}(t_1, *id, k) \\
\text{Tree}(id : t, tr_1, \ldots, tr_m) & \text{if } t = \text{struct} \{i_1 : t_1, \ldots, i_m : t_m\} \\
& \quad \text{and } tr_i = \text{buildTree}(t_i, (id).id_i, k) \\
& \quad \text{for } i = 1 \ldots m \\
\text{buildTree}(\text{TM}(tn), id, k-1) & \text{if } t = tn \text{ and } k > 0 \\
\text{Tree}(id : \text{TM}(tn)) & \text{if } t = tn \text{ and } k = 0 
\end{cases}
\end{align*}

Figure 3.5. Type-based parameter-tree building.

int, a pointer type \(t_1\star\), an anonymous struct type that contains a list of types for the struct’s fields, and a named type with name \(tn\). We use \(tn\) for a type name. We further assume a type map \(TM\), which is a finite map from type names to their type definitions (that is, a collection of typedefs).

For example, the named struct type \(\text{"struct Node \{int value; Node *next\}"}\) is represented as

Fig. 3.5 presents the algorithm for type-based parameter tree building. Given an identifier \(id\) with type \(t\), it builds a tree with root annotated with \(\"id : t\"\) and child trees based on components of \(t\). Notation \(\text{Tree}(id : t, tr_1, \ldots, tr_m)\) is for a tree that has root \(\"id : t\"\) and \(m\) child trees in \(tr_1\) to \(tr_m\). The algorithm in Fig. 3.5 is recursive. If \(t\) is a struct type, it recursively builds subtrees for field types before constructing a tree for the struct type. If \(t\) is a pointer type \(\"t_1\star\"\), we construct a tree of root \(\"id : t\"\) and a subtree based on type \(t_1\) and identifier \(*id\).

Since types may be recursive (as in the case of the \textbf{Node} type), the build-tree algorithm adopts a \(k\)-limiting approach to stop expanding types after \(k\) expansions, avoiding an infinite expansion at the type level. This is implemented in the cases when \(t\) is a type name \(tn\): it decreases \(k\) after expanding the type name using the type map \(TM\) and stops expanding when \(k\) hits zero. Our implementation fixes \(k\) to be one. For the example \textbf{Node} type, the 1-limiting parameter tree is presented in Fig. 3.6.

Semantically, each node in a parameter tree represents an abstract memory
Figure 3.6. Parameter tree for head of type Node *.

region. The type on the node tells the type of elements stored in the memory region. Take the tree of Fig. 3.6 as an example: the root node represents an abstract memory region that holds a sequence of Node* pointers (it is a sequence as head may actually point to an array of Node* elements); the "(*head).next node represents a sequence of Node* pointers as well as all storage those pointers can reach.

A function’s PDG is built as follows: (1) add nodes for instructions in the function; (2) build formal parameter trees for parameters of the function; (3) for a call instruction in the function, build actual parameter trees for arguments of the call; (4) add intra-procedural dependence edges. For a function, we only need to build its formal parameter trees once; by contrast, the actual parameter trees need to be built per function call site.

We next discuss how dependence edges are computed. Intra-procedural dependence edges for a function consists of control-dependence and data-dependence edges. Control dependence can be computed with a classic algorithm [6] based on post-dominator trees.

Def-use data dependence can be computed easily because LLVM IR uses the SSA (Static Single Assignment) form. For a use of a variable in an instruction, it suffices to find the single definition of the variable and add a edge from the definition to the use. A function parameter is conceptually defined at the beginning of the function; therefore, data-dependence edges are added from the root node of a parameter’s tree representation to uses of the parameter.

RAW (Read-After-Write) data dependence is computed with the help of an
intra-procedural pointer analysis. In LLVM IR, only store instructions can write to memory and only load instructions can read from memory. Therefore, for each load instruction, our implementation checks every store instruction in the same function and see whether their destination memory locations can overlap, using the DSA pointer analysis [32]; if so, an edge is added from the store to the load instruction. This construction is flow-insensitive as it ignores the ordering of instructions; it makes an over-approximation.

In addition, we add RAW data-dependence edges between instruction nodes and parameter-tree nodes; examples can be found in Fig. 3.3 (note for succinctness the figure omits RAW labels on data-dependence edges). Conceptually, nodes in a formal parameter tree of a function represent potential reads/writes in the callers of the function; therefore, if the function has a load/store instruction and the instruction accesses memory regions represented by a parameter-tree node, a data-dependence edge should be added between the instruction’s node and the parameter-tree node. Similarly, nodes in an actual parameter tree at a function call site conceptually represent potential reads/writes in the callee function; therefore, data-dependence edges are also added between corresponding instruction nodes and nodes in actual parameter trees.

With parameter trees, inter-procedural dependence representation becomes trivial. For each function call site, we just connect nodes in the actual parameter trees to the corresponding formal parameter trees of the callee function, using bidirectional flow edges.

We note that library function calls (e.g., calls to scanf, printf, exit...) are represented as regular instruction nodes with dependence edges added according to the library functions’ semantics. Alternatively, we could treat library functions as ordinary functions and represent them using PDGs based on their source code, but it would substantially increase the PDG size.

### 3.3.4 PDG-Based Program Partitioning

PtrSplit’s partitioning algorithm takes the PDG of a program and separates it into a sensitive partition with access to sensitive data and an insensitive partition for the rest of the code. The algorithm performs function-level partitioning and does not split a single function. Furthermore, since our PDG represents both data and
control dependence, the algorithm considers both explicit flows of sensitive data (via data dependence) and implicit flows (via control dependence) when deciding what part of code may have access to the sensitive data.

Algorithm 1 PDG-based program partitioning

Input: $G$ is a PDG

Output: $F_s$: the set of sensitive functions; $Gl_s$: the set of sensitive global variables

$sensitive \leftarrow \{n \mid n \text{ is marked sensitive}\}$

$worklist \leftarrow \text{sensitive}$

while $worklist$ is not empty do

$n \leftarrow worklist.pop()$

for data/control dependence edge $n \rightarrow n'$ do

if $n'$ is not declassified and $n' \notin sensitive$ then

$sensitive \leftarrow \{n'\} \cup \text{sensitive}$

$worklist \leftarrow \{n'\} \cup worklist$

$F_s \leftarrow \{f \mid f \text{ has a node } n \text{ in sensitive}\}$

$Gl_s \leftarrow \{g \mid g's \text{ parameter tree has a node } n \text{ in sensitive}\}$

Algorithm 1 presents the PDG-based partitioning algorithm. The input is a PDG and the output is a set of functions $F_s$ and a set of global variables $Gl_s$ that should be put into the sensitive partition; the rest of the program is in the insensitive partition. The $sensitive$ set starts with the set of nodes that programmers mark as sensitive using attributes (an example is line 1 in Fig. 3.2). Then a worklist algorithm is used to compute the set of nodes that a sensitive node can reach along the data-dependence edges (explicit data flow) and the control-dependence edges (implicit data flow) in the PDG, while excluding nodes that programmers mark as declassified nodes (also using attributes). At the end of the algorithm, any function whose PDG contains sensitive nodes is put into the set of sensitive functions and any global variable whose parameter-tree representation contains sensitive nodes is put into the set of sensitive global variables.

For the example PDG in Fig. 3.3, the node with label “*key:char” is marked sensitive. As a result, the $encrypt$ function is sensitive because it has a node with an incoming data-dependence edge from “*key:char”. Similarly, $initkey$ is marked sensitive (its PDG is not shown in Fig. 3.3). Then node “*ciphertext:char” is marked sensitive because of an incoming data-dependence edge. Consequently, $main$ is marked sensitive (because of an edge from “*ciphertext:char” to a node
in main, not shown in Fig. 3.3). In contrast, if node “*ciphertext:char” were marked declassified, then main would not be marked sensitive.

### 3.4 Selective Pointer Bounds Tracking

As discussed before, a core challenge in partitioning C/C++ programs is that pointers do not carry the bounds of the underlying buffers, making marshalling/unmarshalling of pointer data a manual and error-prone process.

Bounds information is also critical for another security application: spatial memory safety. There have been many systems (e.g., [31, 33, 34]) that track bounds information as metadata on buffers or pointers and insert checks before pointer operations to prevent out-of-bound buffer access. However, systems that enforce spatial memory safety incur high performance overhead; e.g., SoftBound’s performance overhead on the SPEC and Olden benchmarks is 67% on average.

For marshalling and unmarshalling it is necessary to perform only bounds tracking, but not bounds checking. That is, it is sufficient to track the bounds of pointers without performing bounds checking on pointer accesses; even if the insensitive partition had an out-of-bound pointer, it would not be able to access the sensitive data through the pointer as it is in a separate process.

We further observe that it is necessary to track the bounds of pointers that can cross the boundary of partitions, but not the bounds of all pointers. Therefore, by performing only bounds tracking for a subset of pointers, the performance overhead should be lower than those systems that enforce spatial memory safety.

Based on this observation, we have designed a Selective Pointer Bounds Tracking (SPBT) system, which (1) computes a set of Bounds-Required (BR) pointers given a partitioning of the program, and (2) instruments the program to track the bounds of those BR pointers.

The algorithm for computing the set of BR pointers is presented in Algorithm 2. It operates on a PDG and takes as input a partitioning of the program, in the form of two sets of functions $F_0$ and $F_1$, one for each partition. The BR set is initialized with the set of pointers that are sent across from one partition to the other partition; obviously, bounds information are required for automatic marshalling and unmarshalling of these pointers.

With a backward propagation process along the data-dependence edges in the
Algorithm 2 Compute a set of BR pointers

Input: \( G \) is the PDG for a program; \( F_0 \) and \( F_1 \) are two disjoint sets of functions that cover the program

Output: \( P \) is the set of bounds-required pointers

\[
\begin{align*}
BR &\leftarrow \emptyset \\
&\text{for function } f \in F_0 \cup F_1 \text{ and call } C \in f \text{ do} \\
&\quad \text{if } (f \in F_i \text{ and } C's \text{ callees } \cap F_{1-i} \neq \emptyset) \text{ then} \\
&\quad \quad \text{for node } n \text{ in } C's \text{ parameter trees do} \\
&\quad \quad \quad \text{if } n's \text{ label is } (id : t*) \text{ then} \\
&\quad \quad \quad \quad BR \leftarrow BR \cup \{(n, id)\}
\end{align*}
\]

\[
\begin{align*}
\text{worklist} &\leftarrow BR \\
\text{while worklist is not empty do} \\
&\quad (n, id) \leftarrow \text{worklist.pop()} \\
&\quad \text{for data-dependence edge } n' \rightarrow n \text{ with label } id_1 \text{ do} \\
&\quad \quad \text{if alias}(id, id_1) \text{ then} \\
&\quad \quad \quad \text{for all pointer-typed } id' \text{ in } n' \text{ do} \\
&\quad \quad \quad \quad \text{if } (n', id') \notin BR \text{ then} \\
&\quad \quad \quad \quad \quad BR \leftarrow BR \cup \{(n', id')\} \\
&\quad \quad \quad \quad \quad \text{worklist} \leftarrow \text{worklist} \cup \{(n', id')\}
\end{align*}
\]

\[
P \leftarrow \{id \mid (n, id) \in BR\}
\]

PDG, the algorithm further computes the set of pointers whose values can flow transitively to the initial BR set. Such pointers also need bounds information because, when a pointer \( p_1 \)'s value flows to \( p_2 \), the bounds of \( p_2 \) is set according to the bounds of \( p_1 \); therefore, if \( p_2 \) is sent over the partition boundary afterwards, \( p_1 \)'s bounds need to be tracked as well. As an example, suppose \( p_1 \) is the result of a memory allocation and its value flows to \( p_2 \), which is sent over the boundary; it is then necessary to create the bounds information for \( p_1 \) at the site of memory allocation and then propagate the information from \( p_1 \) to \( p_2 \).

The algorithm tracks a set of pairs of nodes and identifiers in the sensitive set, instead of a set of nodes. This improves the precision of the algorithm. To illustrate, suppose the PDG has a node \( n \) for instruction “\( p_1 = p_2 + i \)”, where \( p_1 \) is a BR pointer and \( i \) is an integer. The algorithm then puts \((n, p_2)\) into the BR set and performs further processing along \( n \)'s incoming data-dependence edges; during this processing, all edges with label \( i \) can be ignored. Such distinction could not be made if the algorithm used a set of nodes in the BR set.
PtrSplit’s SPBT instrumentation is based on SoftBound [31], an LLVM-based code transformation for enforcing spatial memory safety (another version also enforces temporal memory safety). For each pointer, SoftBound keeps its base and bound. Metadata is created for pointers at allocation sites. Metadata is propagated along with the propagation of pointer values, for example, when pointers are passed during function calls. Finally, before load/store instructions, metadata is used to check for memory-safety violations.

Our SPBT instrumentation removes memory-safety checking before load/store instructions. Furthermore, at an allocation site, if the returned pointer is not in the set of BR pointers (as computed by Algorithm 2), SPBT removes instrumentation that records the pointer’s base and bound metadata. Similarly, when pointer values are propagated, if the involved pointers are not in the set of BR pointers, the instrumentation that propagates metadata is removed.

3.5 Type Based Marshalling and Unmarshalling

Since partitions are loaded into separate processes, some function calls are turned into Remote Procedure Calls (RPCs). During an RPC, arguments from the caller are marshalled into a data buffer and sent to the callee, which unmarshalls the data buffer and recreates the values for the parameters in the callee process. Data marshalling is straightforward for values of most data types, including integers, arrays of fixed sizes, and structs.

However, pointer types cause many troubles. First, C pointers do not carry bounds information; so marshalling does not know the sizes of underlying buffers and cannot marshall the buffers as a result. Second, it is possible to create recursive data structures such as linked lists and arbitrary aliases when using pointers, which makes marshalling/unmarshalling difficult. For instance, if the caller sends a pointer that points to a circular linked list, after marshalling and unmarshalling, a linked list with the same circularity and aliasing should be recreated in the callee process.

Previous program-partitioning frameworks [13, 14] avoid the pointer issue by asking programmers to write manual marshalling and unmarshalling code when pointer data are involved. General Interface Description Languages (IDLs) also do not provide a satisfactory solution. For instance, the Microsoft COM IDL [35] requires manual annotation about the size of a variable-sized array and also an-
notation about aliasing when multiple pointers are involved. The popular SWIG IDL [36] adopts the approach of *opaque pointers*: pointers are wrapped as opaque objects and are sent over the boundary without copying the underlying buffers; whenever the callee domain needs to perform operations on those pointers, the control is transferred back to the caller domain for the actual operations. The opaque-pointer approach avoids the pointer issue, but it may lead to frequent domain crossings; further, it may cause a security problem if an untrusted partition can spoof opaque pointers to read arbitrary memory; some solution for opaque pointer integrity would be needed. Finally, popular RPC libraries (e.g., Google’s gRPC [37] and Oracle’s TI-RPC [38]) also do not provide good support for pointers and require manual intervention.

By applying SPBT, all pointers that cross the partition boundary in our system are equipped with bounds information, making it possible to automatically marshall/unmarshall even pointer data. Therefore, we propose the approach of *type-based deep copy* of pointer data: a pointer value is marshalled along with the buffer the pointer points to; if the buffer itself contains pointers, those pointers are marshalled recursively; the callee process unmarshalls the received data to recreate the pointer and the buffer, also in a recursive way; furthermore, as we will discuss, our approach takes care of circularity and aliasing in data.

We next present a formal algorithm for type-based marshalling/unmarshalling that performs deep copying of pointer data. We will use the same set of types in Fig. 3.4 when presenting the algorithm. In addition, the syntax for values is as follows:

\[
Value \ v := n \ | \ struct \ \{id_1 = v_1, \ldots, id_n = v_n\} \ | \ p(b,e)
\]

A value can be an integer \(n\), a struct value with field values inside, or a pointer value of the form \(p(b,e)\). After SPBT, all pointers that cross the boundary have bounds information in the form of \((b,e)\), where \(b\) is the beginning of the underlying buffer, \(e\) is the end of the buffer, and the buffer size is \(e - b\). A null pointer is encoded as \(0(0,0)\) (that is, it points to an empty buffer).

Fig. 3.7 presents a recursive algorithm for encoding a value \(v\) of type \(t\) into a list of bytes. In the figure, we use \([\ ]\) for an empty list, and \(l_1 + l_2\) for the concatenation of two lists. The algorithm assumes a list of utility functions, which are explained in the caption.

The case when \(t = \text{int}\) is simple; just encode the type and the integer. For a
\[
\text{encode}(B, t) = \begin{cases} 
\text{enc\_typ}(t) + \text{enc\_int}(n) & \text{if } t = \text{int} \text{ and } v = n \\
\text{enc\_typ}(t) + l_1 + \ldots + l_n & \text{if } t = \text{struct } \{ id_1 : t_1, \ldots, id_n : t_n \} \\
\text{enc\_typ}(t) + \text{enc\_ptr}(p(b,e)) + l_{buf} & \text{if } t = t_1 \ast \text{ and } v = p(b,e) \\
\text{encode}(B, \text{TM}(tn)) & \text{if } t = tn
\end{cases}
\]

\[
\text{enc\_buf}_{B}(b, e, t) = \begin{cases} 
\text{encode}(B, t) + \text{enc\_buf}_{B}(b + \text{size}(t), e, t) & \text{if } b + \text{size}(t) \leq e \text{ and } v = \text{read\_mem}(b, t) \\
[] & \text{otherwise}
\end{cases}
\]

\[\text{Figure 3.7.} \text{ Type-based marshalling. In the algorithm, we assume a set of basic utility functions: enc\_typ}(t) \text{ for encoding a type into a list of bytes; enc\_int}(n) \text{ for encoding an integer; enc\_ptr}(p(b,e)) \text{ for encoding a pointer; size}(t) \text{ for the size of values in type } t; \text{ read\_mem}(b, t) \text{ for reading a value of type } t \text{ from memory at address } b.\]

struct type, all field values and their types are encoded. For a named type \(tn\), the value is encoded according to the type definition for \(tn\) as defined in the type map \(\text{TM}\).

The case for a pointer type is challenging since the algorithm has to deal with circularity caused by pointers. For that, the encode function also takes a parameter \(B\), which remembers a list of buffers (in the form of \((b,e)\)) that have already been encoded; when encoding a pointer that points to an already encoded buffer, there is no need to encode the buffer again. If the buffer has not been encoded, \((b,e)\) is added to \(B\) and every element in the buffer is then encoded recursively (with the help of function \(\text{enc\_buf}_{B}(b, e, t)\)).

As an example, assume we need to marshall a circular linked list of two nodes, shown in Fig. 3.8. Each node is of type \texttt{Node} with two fields, one is type int and one \texttt{Node}\*; each field is assumed to occupy four bytes. To marshall this data structure, we make the following call:

\[
\text{encode}_{\emptyset}(p_1(p_1, p_1+8), \texttt{Node}\*)
\]

This call encodes the \(p_1\) pointer as well as the buffer it points to; the buffer contains the first node (viewed as an array of one node). When encoding the buffer, because of the pointer inside the first node, the encoder is recursively invoked as follows:

\[
\text{encode}_{\{p_1(p_1+8)\}}(p_2(p_2, p_2+8), \texttt{Node}\*)
\]
At this point, however, only the pointer is encoded, not the underlying buffer since function has an additional parameter in the sender partition and buffers in the receiver partition. This is why the decode function has an additional parameter M for remembering the map. There are two cases, for pointer p(b,e) that is sent, if (b,e) is not recorded in M, then the receiver has not allocated an corresponding buffer yet; in this case, a new buffer is...
allocated and initialized by the `dec_buf` function. If \((b, e)\) has been recorded in \(M\), then the corresponding buffer has already been allocated and there is no need for reallocation. In both cases, the returned pointer value uses the bounds information of the buffer in the receiver and \(p\) is adjusted to be \(b' + p - b\) to maintain the offset between the pointer and the beginning of the buffer.

For the example of circular linked lists, the decoder allocates a node for each node in the original linked list and at the same time adjusts pointer values according to the buffer map \(M\).

The previous algorithm shows how to marshall/unmarshall one argument, but our implementation marshalls and unmarshalls all arguments at the same time. This is important, not just for efficiency, but for correctness in the case when multiple pointer arguments alias the same buffer; the buffer should be encoded just once so that the receiver can recreate aliasing. Essentially, this approach treats multiple arguments as a value of a tuple type.

When a pointer is passed from a caller to a callee partition, `PtrSplit` performs deep copying of pointer data. Since the callee may modify such data, it is necessary to copy back the entire pointer data from the callee and caller at the end of the RPC call. This implements the copy-in and copy-out semantics for pointer data, which is compatible with single-threaded code.

After marshalling, arguments of a function call are encoded as a byte array, which is sent to the receiver via the help of an RPC library. We use the popular TI-RPC library [38] for sending and receiving byte arrays.

In our system, deep copying of pointer data applies to only user-space data pointers. Our implementation maintains a whitelist of other kinds of pointers that are not deep copied, including pointers to OS-kernel data structures and pointers to code. It is not possible to deep copy these pointers; therefore we adopt the opaque-pointer approach for them. For instance, when one partition creates a file pointer through the OS, our marshalling wraps the file pointer as an opaque object without performing deep copying. The receiver is transformed to send the file pointer back to the sender for operating on the underlying file. For a code pointer that crosses the boundary, our system also wraps it as an opaque pointer with a runtime tag; an indirect call via a code pointer is instrumented to decide whether the code pointer is local or remote before performing a local or an RPC call.
Table 3.1. Partitioning results of security-sensitive programs. (Abbreviations: "Total
pointers": total pointer variables in LLVM-IR; "BR pointers": bounds-required pointer
variables; "PBT": pointer bounds tracking; "SPBT": selective pointer bounds tracking.)

<table>
<thead>
<tr>
<th></th>
<th>SLOC</th>
<th># of functions/ sensitive functions</th>
<th>Total/BR pointers</th>
<th>PBT overhead</th>
<th>SPBT overhead</th>
<th>Total overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssh</td>
<td>64,671</td>
<td>1235/12</td>
<td>21020/591</td>
<td>45.0%</td>
<td>2.6%</td>
<td>7.4%</td>
</tr>
<tr>
<td>wget</td>
<td>61,216</td>
<td>666/8</td>
<td>14939/466</td>
<td>52.5%</td>
<td>3.4%</td>
<td>6.5%</td>
</tr>
<tr>
<td>thttpd</td>
<td>21,925</td>
<td>145/5</td>
<td>3068/189</td>
<td>56.3%</td>
<td>3.6%</td>
<td>8.8%</td>
</tr>
<tr>
<td>telnet</td>
<td>11,118</td>
<td>180/11</td>
<td>2068/233</td>
<td>74.1%</td>
<td>5.1%</td>
<td>9.6%</td>
</tr>
</tbody>
</table>

3.6 Evaluation

We implemented PtrSplit in LLVM 3.5. SoftBound's public release only supports
LLVM 3.4; so we had to upgrade its code base to support LLVM 3.5. Several
LLVM passes were added to implement the components of PtrSplit. We evaluated
PtrSplit using a set of benchmarks on a system running x86-64 Ubuntu 14.04 with
the Linux kernel version 3.19.0, an Intel Core i5-4590 at 3.3GHz, and 16GB of
physical memory.

The evaluation aims to answer several questions: (1) whether PtrSplit can
automatically partition realistic C applications and scale to relatively large C
applications, (2) whether the performance overhead of a partitioned application
is acceptable, given the overhead of performing SPBT and deep copying of RPC
data, (3) whether SPBT significantly reduces the overhead, when compared with a
solution that enforces full spatial memory safety.

We first evaluated with a set of microbenchmarks to validate the major func-
tionalities of PtrSplit. The programs include the running example in Fig. 3.2 and
programs that send data structures (including trees, linked lists, and circular linked
lists) over RPC calls.

We then evaluated PtrSplit with a set of security-sensitive programs and pro-
grams from SPECCPU 2006. For each program, we ran its partitioned version and
checked that the partitioned version functioned well using the reference data set
attached with the program. During the process, we also measured the performance
overhead of the partitioned version. These experiments are detailed next.

We evaluated PtrSplit on four security-sensitive programs. Considering that
all of these programs are networking programs, which are greatly affected by the
network latency, we used another machine that was in the same LAN as a remote
server. The remote server machine has the same hardware and OS configuration as the local machine. For each program, we analyzed its functionality and marked some sensitive data that need isolation; recall that sensitive data means data of either high secrecy or low integrity. Then PtrSplit is used to partition these programs to isolate sensitive code and data into a separate partition. Results for these programs are presented in Table 1. We next discuss in detail how experiments were performed for each program.

\textbf{ssh} is a networking utility included in \texttt{OpenSSH} (version: 7.4p1), which is a suite of utilities based on the SSH protocol. The \texttt{ssh} utility implements the client-side of the SSH protocol. We annotated the buffer that receives the RSA private key as the sensitive data. We also declassified the return results of functions \texttt{sshkey\_load\_file} and \texttt{sshkey\_load\_private}; although these functions compute on sensitive data, their return results are status/error codes, which are not correlated to sensitive data. (The reason for declassification in \texttt{wget} and \texttt{telnet} is the same and we will not repeat it when we discuss those programs.) In total, twelve functions were put into a sensitive partition. For measuring performance overhead, we used our partitioned \texttt{ssh} to log in to the server one hundred times.

\texttt{wget} (version: 1.18) is a command-line program for retrieving files from a remote HTTP or FTP server. We annotated the buffer for receiving the downloaded file from an FTP server as the sensitive data because the file may contain malicious content. We also declassified the return results of functions \texttt{fd\_read\_body} and \texttt{skip\_short\_body}. For measuring performance overhead, we downloaded a 1KB file from the FTP server one hundred times.

\texttt{thttpd} (version: 2.27) is an open-source http server program. We chose its authentication file as the sensitive data, and annotated the corresponding buffer that reads contents from the authentication file in the source code; a single declassification annotation was also added to declassify the result of function \texttt{auth\_check}. After separation, five functions that access the authentication-file buffer were put into a sensitive partition. To measure the average overhead, we set up a server on the remote machine with our partitioned \texttt{thttpd} and downloaded a 1KB file on that server multiple times through a local client.

\texttt{telnet} (version: inetutils-1.9.4) is a networking client utility based on the telnet protocol. We consider the threat of a remote entity that pretends to be a server and the client somehow connects to the fake server (in a phishing attack, e.g.)
and the fake server tries to use a vulnerability to attack the client. To counter the threat, we annotated the buffer that receives packets from the server as the sensitive data because the received packets may contain malicious content. We also declassified the return results of functions `telrcv`, `ttyflush` and `process_rings`. In total, eleven functions were put into a sensitive partition. We measured the average performance overhead of using our partitioned telnet to log in to a remote server one hundred times.

Overall, our experiments showed promising results, shown in Table 3.1. For each program, the table lists its lines of source code, the sensitive data, the total number of functions in the program versus the number of functions in the sensitive partition computed by the partitioning algorithm, the total number of pointers (i.e., static counts of pointer variables) versus the number of Bounds-Required (BR) pointers computed by SPBT, the performance overhead (compared to the vanilla, uninstrumented program) when full pointer bounds tracking is applied, the performance overhead when only SPBT is applied, and the total performance overhead for the partitioned application.

As shown in the table, SPBT is effective at reducing the overhead of pointer bounds tracking and the overall performance overhead of the security-sensitive applications is acceptable. They demonstrated that PtrSplit can be used for partitioning realistic security-sensitive applications to improve security, with a modest amount of performance overhead.

We then evaluated PtrSplit using the SPECCPU 2006 C benchmarks. These programs are compute-intensive benchmarks and are not security-sensitive benchmarks. However, we felt it is important to evaluate PtrSplit using compute-intensive benchmarks as they stress test the instrumentation mechanism of PtrSplit; furthermore, we would like to compare the performance overhead of SPBT with the overhead of full pointer bounds tracking (PBT) on SPEC benchmarks. For each of the benchmarks, we randomly selected a global variable, marked it sensitive, and fed it to PtrSplit; in this experiment, only explicit flows are taken into account and no declassification is used during partitioning since it is not for evaluating security but for evaluating the instrumentation mechanism.

Table 3.2 presents the experimental results. We note that three programs (`perlbench`, `gcc`, and `gobmk`) were excluded because SoftBound’s memory-safety instrumentation produces runtime crashes due to SoftBound’s implementation bugs.
### Table 3.2. Partitioning results for SPECCPU 2006 benchmarks (use a random global as sensitive variable).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SLOC</th>
<th>Sensitive data and type</th>
<th># of functions/ sensitive functions</th>
<th>Total/BR pointers</th>
<th>PBT</th>
<th>SPBT</th>
<th>Total overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>lbm</td>
<td>1,156</td>
<td>LBM_Grid* srcGrid</td>
<td>19/5</td>
<td>695/131</td>
<td>141.4%</td>
<td>19.7%</td>
<td>24.3%</td>
</tr>
<tr>
<td>libquantum</td>
<td>4,358</td>
<td>struct quantum_reg* lambda</td>
<td>115/3</td>
<td>1690/128</td>
<td>282.3%</td>
<td>11.2%</td>
<td>179.2%</td>
</tr>
<tr>
<td>bzip2</td>
<td>8,393</td>
<td>char* progName</td>
<td>100/6</td>
<td>4356/8</td>
<td>59.4%</td>
<td>3.1%</td>
<td>5.3%</td>
</tr>
<tr>
<td>sjeng</td>
<td>13,547</td>
<td>char* realholdings</td>
<td>144/5</td>
<td>3415/81</td>
<td>41.7%</td>
<td>3.4%</td>
<td>10.2%</td>
</tr>
<tr>
<td>milc</td>
<td>15,042</td>
<td>double[] path_coeff</td>
<td>235/2</td>
<td>5001/0</td>
<td>111%</td>
<td>0%</td>
<td>2.2%</td>
</tr>
<tr>
<td>sphinx3</td>
<td>25,090</td>
<td>char** liveargs</td>
<td>369/3</td>
<td>9491/37</td>
<td>90.5%</td>
<td>5.1%</td>
<td>7.1%</td>
</tr>
<tr>
<td>hmmer</td>
<td>35,992</td>
<td>int ser_randseed</td>
<td>538/7</td>
<td>17692/175</td>
<td>128.5%</td>
<td>5.8%</td>
<td>26.7%</td>
</tr>
<tr>
<td>h264ref</td>
<td>51,578</td>
<td>int[] FirstMBInSlice</td>
<td>590/5</td>
<td>32212/461</td>
<td>234.4%</td>
<td>9.6%</td>
<td>15.5%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>136.2%</strong></td>
<td><strong>7.2%</strong></td>
<td><strong>33.8%</strong></td>
</tr>
</tbody>
</table>
and PtrSplit’s SPBT implementation is on top of SoftBound. The original Soft-
Bound paper also did not report results on perlbench and gcc; further, paper [39] reported the difficulty of instrumenting SPEC benchmarks using SoftBound. We also excluded mcf because it is a small program with 24 functions and any global variable marked as sensitive would lead to all functions being in one partition (adding declassification annotations would produce a separation, but we refrained from doing so since it is unclear where to declassify based on a randomly selected global variable).

For SPBT, we can see from the table the total number of pointers that require bounds is typically a small percentage of the total number of pointers in a program (we counted the number of pointers statically, based on their types). As a result, the average SPBT runtime overhead for the benchmarks is 7.2%, which is much lower than the average overhead of 136.2% for full pointer bounds tracking (PBT). This shows the effectiveness of SPBT. Note that milc has no BR pointers because no pointer data are passed between the created partitions.

The runtime overhead of PtrSplit comes from two sources: pointer bounds tracking and data marshalling/unmarshalling for RPC calls. Table 3.2 also shows the total runtime overhead. libquantum’s overhead is rather large; we found that RPC call overhead is positively correlated to the RPC call frequency. For libquantum, the randomly selected variable leads to a partitioning with a high RPC call frequency (94 Hz); the RPC call frequency of other benchmarks is below 3Hz. Choosing a different global variable of libquantum would lead to a similar result.

Table 3.3. The random partitioning results for SPECCPU 2006 benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Average BR-pointer ratio</th>
<th>Average SPBT overhead</th>
<th>Average total overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>lbm</td>
<td>14.3%</td>
<td>15.4%</td>
<td>55.1%</td>
</tr>
<tr>
<td>libquantum</td>
<td>16.2%</td>
<td>51.5%</td>
<td>163.3%</td>
</tr>
<tr>
<td>bzip2</td>
<td>12.4%</td>
<td>16.4%</td>
<td>71.3%</td>
</tr>
<tr>
<td>sjeng</td>
<td>15.2%</td>
<td>14.1%</td>
<td>63.9%</td>
</tr>
<tr>
<td>milc</td>
<td>10.7%</td>
<td>23.4%</td>
<td>83.2%</td>
</tr>
<tr>
<td>sphinx</td>
<td>8.7%</td>
<td>17.9%</td>
<td>37.5%</td>
</tr>
<tr>
<td>hmer</td>
<td>8.8%</td>
<td>29.8%</td>
<td>89.7%</td>
</tr>
<tr>
<td>h264ref</td>
<td>9.1%</td>
<td>38.4%</td>
<td>101.9%</td>
</tr>
<tr>
<td>Average</td>
<td>11.8%</td>
<td>29.4%</td>
<td>79.3%</td>
</tr>
</tbody>
</table>
To further validate the robustness of our partitioning framework, for each SPEC benchmark, we built a script that randomly splits the benchmark’s set of functions into two disjoint sets of functions and creates a partitioning based on the split. The script was run multiple times and for each run we checked that the partitioned application worked as intended (using the reference data set included in SPECCPU 2006). Some of these random partitionings created complex interfaces that required exchange of complex data (structs, pointers, etc.) and provided good stress tests of PtrSplit’s RPC mechanism. Table 3.3 presents the results. For each benchmark, the table includes the BR-pointer ratio (the number of BR pointers divided by the number of total pointers), the SPBT overhead, and the total overhead, averaged over multiple runs of performing random partitioning. The total overhead is on the high side, which indicates random partitioning would not lead to efficient partitionings.
Chapter 4  |  Program-mandering: Quantitative Privilege Separation

Privilege separation is an effective technique to improve software security. However, past partitioning systems (including PtrSplit) do not allow programmers to make quantitative tradeoffs between security and performance. In this chapter, we describe our toolchain called Program-mandering (PM). It can automatically find the optimal boundary in program partitioning. This is achieved by solving an integer-programming model that optimizes for a user-chosen metric while satisfying the remaining security and performance constraints on other metrics. We choose security metrics to reason about how well computed partitions enforce information flow control to: (1) protect the program from low-integrity inputs or (2) prevent leakage of program secrets. As a result, functions in the sensitive module that fall on the optimal partition boundaries automatically identify where declassification is necessary. We used PM to experiment on a set of real-world programs to protect confidentiality and integrity; results show that, with moderate user guidance, PM can find partitions that have better balance between security and performance than partitions found by PtrSplit, which requires manual declassification.

4.1 Summary

PM enables quantitative tradeoffs between security and performance, while achieving meaningful security goals. In particular, PM makes the following contributions:

- PM is a privilege-separation framework that guides users to make quanti-
tative tradeoffs between security and performance. By converting privilege separation into an integer-programming problem, it automatically computes the optimal partition, with respect to user-specified budgets on security and performance.

- PM is the first system that combines quantitative information flow with privilege separation. This not only provides a security metric that aligns well with security goals common in applications, but also reduces users’ burden of performing manual declassification—the optimal partition computed by PM automatically gives where data should be declassified.

- We have implemented PM and evaluated it on a set of real world programs. Our experience shows that PM helps users make quantitative trade-offs among multiple factors. After observing initial partitions, users could use PM to improve the balance between security and performance by setting simple constraints, in an iterative process.

4.2 A Motivating Example

Figure 4.1 presents a toy program motivating the need for balancing security and performance when performing privilege separation. For brevity, we do not involve global variables in the toy program. The program is a simplified version of how the `httpd` web server performs authentication. It accepts a username and a password from the user and performs authentication by using a password file. The password file can be any one of the five possibilities in the `fname` array. The `auth` function iterates over the five possibilities and invokes `auth2`, which checks if the password file exists and, if so, performs authentication by comparing the user name and password string with lines in the password file. In the worst case, `auth` invokes `auth2` five times. Note that the `main` function has a vulnerability that can be used to cause a buffer overrun. As a result, when all three functions are in the same protection domain, an attacker can use the buffer overrun to take over the program and learn information in the password file.

For better security, one partition is to put `auth2` in its own protection domain, with the privilege of reading the password files, and the rest of the code stays in a different protection domain, which lacks access to any password file. As a result,
Figure 4.1. A motivating example.

less code has access to the secret passwords. While this is a natural partition for improving security, other choices might be better to have better balance between security and performance.

To explore possible choices of partitioning, let us examine the example’s call graph annotated with information about frequencies of calls (represented by the numbers of calls for brevity) and potential leakage of secret information, shown in Figure 4.2. It shows that main invokes auth once and auth potentially invokes auth2 five times. The return value of auth2 potentially leaks one bit of the secret password since its return value depends on the comparison between the user-provided password and the real password. That one bit is also propagated back from auth to main, since auth’s return value depends on the result of calling auth2.

Figure 4.2 also shows two possible partitions. Partition #1 is the one we
have already discussed. It minimizes the size of the sensitive domain (assuming functions are the unit of partitioning). To produce partition #1 in Ptrsplit, one would manually declassify the return values of auth2. Partition #2 puts auth and auth2 in the sensitive domain and main in the insensitive domain. Partition #2 would likely cause less runtime overhead than #1 because it has fewer cross-boundary function calls (1 versus 5 calls), at the cost of an enlarged sensitive domain. In addition, less data must be transferred between partitions, we send only the requested password for partition #2, not the password file name as well. Therefore, partition #2 may be more desired depending on how one considers different factors. In Ptrsplit, partition #2 can be achieved by manually declassify the return result from auth to main.

This example shows that the best partition highly depends on what the users’ tradeoffs are among multiple factors. Ptrsplit resorts to asking users to analyze the situation and use manual declassification and code duplication to find a good partition. However, such manual analysis is laborious and error-prone. Therefore, what is needed is a flexible, automatic framework that outputs the best partition according to users’ requirements.

### 4.3 System Overview

Figure 4.3 presents PM’s workflow. It takes the source code of an application as input and constructs a Program Dependence Graph (PDG) for the application. The user also annotates the application to tell PM what the sensitive data is. PM then uses program analysis to get measurements on partitioning factors, including security, performance, and interface complexity.
For security, PM uses a dynamic information-flow tracker to measure the quantity of sensitive flow among functions and global variables during runtime. In addition, PM records the code sizes of functions to compute the code percentage of the sensitive partition in a partitioning; a smaller sensitive partition means a smaller trusted computing base and better security. For performance, a profiler is run to collect how frequently functions are called and global variables are accessed, as a way of approximating the overhead caused by context switches caused by a partitioning. For interface complexity, static analysis is employed to compute the type complexity of function interfaces in terms of levels of pointers.

Those measurements are used to annotate the nodes and edges of the PDG. Based on the annotated PDG and user-specified constraints on factors such as security, a partitioning algorithm searches for a partitioning that satisfies the constraints and is optimal according to a user-specified goal. The output is a sensitive partition and an insensitive partition, each of which consists of both data and code.

We next clarify a few points. First, user-specified constraints restrict the search space of what partitions are acceptable to users. For instance, a user can specify that the sensitive information flow from the sensitive domain to the insensitive one should be at most 2 bits. It can be difficult to get those constraints right in one shot; so PM is intended as an interactive tool for users. A user specifies some initial constraints and chooses the metric to optimize and PM computes the optimal partition for that optimization metric under those constraints; then the user inspects the results and possibly makes adjustments to the constraints to get further partitions. Second, PM’s partitioning granularity is at the function level; it does not partition individual functions. Our experiments show that this level

Figure 4.3. System flow of PM.
of granularity is sufficient for many programs; however, there are some programs whose partitioning would require finer granularity, as we will discuss.

4.4 Graph-Based Partitioning

We next formalize program partitioning as graph partitioning. We then show how we can encode the problem in integer programming.

4.4.1 Graph partitioning

When partitioning a program with both functions and global variables, PM splits the program into two domains, each with a set of functions and globals. For each global, a getter function and a setter function are added. A domain can access its own globals directly; however, to access a different domain’s globals, RPCs are issued to the getter/setter functions. As a further optimization, PM determines what global data are read only and duplicates all read-only global data in both domains, reducing the frequency of context switches caused by accessing globals. Given the above understanding, Appendix 1 presents necessary adjustments to graph partitioning when the input program has both functions and globals. Briefly, the graph becomes a Program Dependence Graph (PDG). In this PDG, vertices represent either functions or globals; edges are either call edges between functions, or data-flow edges between functions and globals. Weights are added on vertices and edges. Note that our implementation reuses PtrSplit to construct full-fledged PDGs [6], which in addition contain control-dependence edges and data-dependence edges within functions. PM, however, needs only the PDG’s call-graph part as well as the data-dependence edges between functions and globals, since it performs function-level partitioning.

Definition 1 (Partitions). A partition of $G = (V, E)$, also called a cut, is modeled as two sets of functions $(S, T)$: a sensitive domain $S \subseteq V$ and an insensitive domain $T \subseteq V$ and they satisfy (1) $S$ contains the special function (i.e., $s \in S$), and (2) $S \cup T = V$.

Note that our partitions allow function replication; that is, $S \cap T$ may not be empty because there may be functions that are replicated in both domains. We
use \( R = S \cap T \) for the set of replicated functions. In practice, duplicating common utility functions often benefits performance. For instance, \texttt{thttpd} has a function called \texttt{my\_snprintf}, a custom implementation of \texttt{snprintf}. It is called by many functions; without duplication, it would cause many domain crossings no matter what domain \texttt{my\_snprintf} would be in. Duplicating it in both domains benefits performance, at the cost of larger domains.

The edges between two domains are called boundary edges; we write \( BE \) to represent the set of boundary edges. There are two kinds of boundary edges. Forward boundary edges are those from \( S \) to \( T - R \), defined as \( FB = \{ e_{ij} \mid i \in S \land j \in T - R \} \). Backward boundary edges are those from \( T \) to \( S - R \), defined as \( BB = \{ e_{ij} \mid i \in T \land j \in S - R \} \). We have \( BE = FB \cup BB \). Note that self recursion does not pose a problem to our formalization: if there is a call edge from \( f \) to itself, it is not considered a boundary edge by the definitions.

Given a call graph, PM annotates its vertices and edges with a set of weights, which represent measurements of security and performance impact at the function level. For example, a function vertex may be associated with a weight that specifies the code size of the function, describing one aspect of the function’s impact on the security of a partition based on the amount of code in a sensitive domain that includes this function. Given a partition, weights in the graph are used to compute metrics for the partition. PM is largely independent of the choice of metrics, but we describe the metrics chosen in our experiments in Section 4.5. We discuss in Sec. 4.5.2 about the space of metrics and how PM can be switched to other metrics.

Given an annotated graph, users guide PM’s partitioning process by specifying constraints and an optimization goal. Constraints are in the form of budgets on metrics: \( b_i \in B \), where \( b_i \) is a limit for the value of metric \( m_i \in M \), \( B \) is the set of budgets, and \( M \) is the set of metrics. The optimization goal is one of the metrics. PM’s goal is to search for the optimal partition in the following sense:

**Definition 2** (Optimal partitioning). For a set of metrics \( M \), a target metric \( m_k \) to minimize, and budgets \( B \), the optimal partition \( P = (S, T) \) is the one that minimizes the target metric and satisfies the following constraint: \( \forall m_i \in M, m_i(P) \leq b_i \), where \( m_i(P) \) is the value of metric \( m_i \) for partition \( P \).

**Global variables.** When partitioning a program with both functions and global variables, PM splits the program into two domains, each with a set of functions and
globals. For each global, a getter function and a setter function are added. A domain can access its own globals directly; however, to access a different domain’s globals, RPCs are issued to the getter/setter functions. As a further optimization, PM determines what global data are read only and duplicates all read-only global data in both domains, reducing the frequency of context switches caused by accessing globals. Note that our implementation reuses PtrSplit to construct full-fledged PDGs [6], which in addition contain control-dependence edges and data-dependence edges within functions. PM, however, needs only the PDG’s call-graph part as well as the data-dependence edges between functions and globals, since it performs function-level partitioning.

### 4.4.2 Partitioning with integer programming

Given the formalization above, we now discuss how to find the optimal partition using binary Integer Programming (IP). IP is linear programming with only integral variables. Solving IP problems in general is NP-complete, but practical IP solvers have been developed and can solve moderately sized IP problems. A binary IP problem is a special IP problem in which all variables are either 0 or 1. To formulate an IP problem, one first needs to declare integral variables with constraints. Constraints can be linear equations or inequations on variables. Afterwards, one must define an objective function to optimize. A solution to an IP problem is an assignment to all declared variables that satisfies all the constraints and optimizes the objective function.

Appendix 2 presents in detail how to encode optimal partition as a binary IP problem for the metrics we introduce in Sec 4.5. Briefly, we introduce binary variables that model (1) whether a function (or a global variable) is only in the sensitive domain (or only in the insensitive domain), and (2) whether an edge is a forward boundary edge (or a backward boundary edge). Then constraints are added to (1) allow only valid partitions (e.g., the special sensitive function must be in the sensitive domain), and (2) limit a produced partition to respect the given budgets. Finally, an objective function is formalized to minimize the target metric. Overall, this encoding declares $O(|E|)$ number of variables and constraints.
4.5 Metrics

4.5.1 Definition of metrics

We define a set of metrics for quantifying the quality of a partition in terms its security and performance. Based on the discussion above, we introduce two metrics for measuring the security impact of a partition: (1) the amount of sensitive information that flows from the sensitive to the insensitive domain and (2) the percentage of sensitive code. For performance, we define metrics to estimate the performance overhead created by the partition using: (1) the frequency of context switches (i.e., the frequency of domain crossings between the two domains) and (2) the pointer complexity of the interface between the two domains (i.e., the amount of data conveyed on domain crossings).

An edge is annotated with information-flow measurements. Recall that an edge $e$ represents a call relation between a caller and a callee. For information flow, two weights are added to an edge: for an edge $e$ that represents calls from $f_1$ to $f_2$, $\text{fflow}(e)$ is the amount of sensitive information, measured in the number of bits, in the arguments passed from $f_1$ to $f_2$; and $\text{bflow}(e)$ is the amount of sensitive information in return values from $f_2$ to $f_1$. We will later discuss how PM uses a dynamic information-flow tracker to measure $\text{fflow}(e)$ and $\text{bflow}(e)$. Then the total amount of sensitive information flow from the sensitive to the insensitive domain is the sum of forward flows on forward boundary edges and backward flows on backward boundary edges:

**Definition 3** (Sensitive information flow).

\[
\text{Flow}(S, T) = \sum_{e \in FB} \text{fflow}(e) + \sum_{e \in BB} \text{bflow}(e).
\]

Each vertex is annotated with the code size of the corresponding function. We write $\text{sz}(v)$ for the code size of the function represented by vertex $v$. Therefore, the total code size of the sensitive domain in a partition $P = (S, T)$ is defined as the sum of the sizes of functions in the domain:
Definition 4 (Sensitive code percentage).

\[
\text{SCode}(S, T) = \frac{\sum_{v \in S} \text{sz}(v)}{\sum_{v \in S \cup T} \text{sz}(v)}.
\]

An edge from \(f_1\) to \(f_2\) is also annotated with a weight \(af(e)\) (abbreviation for access frequency) for measuring the frequency of calls of \(f_2\) by \(f_1\). When \(e\) is a boundary edge, \(af(e)\) corresponds to the frequency of context switches caused by realizing \(e\) as an RPC. Then the total frequency of context switches caused by a partition is the sum of access frequency on boundary edges:

Definition 5 (Context switch frequency).

\[
\text{CSwitch}(S, T) = \sum_{e \in FB \cup BB} af(e).
\]

We also propose a metric for the cost per switch. However, estimating that cost accurately is difficult because it depends on what data is passed and how the context switch is implemented. RPCs are implemented by marshalling the arguments from the caller to the callee, who unmarshalls these values, executes the operation, and performs the reverse process for return values. While data of non-pointer types can be automatically marshalled, more cost is incurred when marshalling C/C++ style pointer data. For example, PtrSplit [5] tracks the buffer bounds of pointers at runtime and copies the underlying buffers during marshalling, causing more data to cross the boundary. In addition, pointers to user-defined types may further reference pointers to other types, possibly necessitating a “deep copy” to convey the necessary data between domains. Alternatively, one may employ the opaque-pointer approach (e.g., used in [36]); however, passing multi-level pointers across the partition boundary opaquely creates frequent domain crossings, since each time an opaque pointer is used the pointer has to be passed back to the sender for processing. Therefore, in either deep copying or opaque pointers, pointer types create significant cost. Given the lack of analyses to estimate RPC overhead accurately at present, we propose a coarse estimate of this overhead based on the pointer complexity of the type signature of the callee function. In this dissertation, the pointer complexity of a type is defined by the level of pointers in the type. For example, the pointer complexity of a base type like \(\text{int}\) is 0, the complexity of \(\text{int}^{*}\) is 2, and the complexity of the type of pointers to structs with a two-level
pointer field is 3.

Given an edge $e$ from $f_1$ to $f_2$ in the call graph, we use $\text{plevel}(e)$ for the sum of pointer complexity of the argument and return types of $f_2$. Then the pointer complexity is defined as the sum of type complexity of all boundary edges:

**Definition 6** (Pointer complexity).

$$\text{Cplx}(S, T) = \sum_{e \in FB \cup BB} \text{plevel}(e).$$

### 4.5.2 Alternative metrics

We next briefly discuss alternatives to our proposed metrics and how PM could be changed to incorporate those metrics. For software-security metrics, we emphasize that the research is lacking, and there are no generally agreed-upon metrics for measuring software security. One possibility is to measure the number of past known vulnerabilities that can be mitigated through partitioning. We did not use this because this metric reflects only the past and does not consider unknown vulnerabilities. Another possibility is to measure the attack surface after partitioning, but how to perform such a measurement is an open question. For performance metrics, there are many other alternatives. In addition to context-switch frequency and pointer complexity, one can dynamically measure the amount of data passed for a cross-boundary call or return, or statically compute the type complexity of parameters (not just for pointers). In this dissertation, we do not attempt to design new metrics, but reuse existing metrics and we find that the metrics we propose are reasonable proxies of security and performance. A follow-up study can investigate which metrics are most appropriate by evaluating metrics on a larger number of benchmarks.

We note that PM is set up in a way that enables users to switch to other metrics without changing its optimization framework for finding the best partition according to a user-specified goal. A new metric can be added as long as measurements at the PDG node/edge level can be performed and the computation of going from node/edge level measurements to partition-level measurements can be encoded in IP. We believe most metrics satisfy these requirements. As an example, for the number of past-known vulnerabilities, we can annotate each function node with the number of vulnerabilities that have been discovered in that function; the number of
vulnerabilities mitigated through isolating an untrusted domain is just the sum of
numbers on the function nodes in the domain, which can be easily encoded in IP.

4.6 Implementation

The prototype implementation of PM is implemented with the help of several tools,
including LLVM\textsuperscript{1}, Flowcheck [41], Intel’s Pin tool [43], and lp\_solve [42]. We first
explain the toolchain’s input and output, followed by a discussion of how each
component is implemented.

\textbf{Tool input and output.} At a high level, PM’s implementation takes two pieces
of input from the user: (1) source code plus user annotations about what sensitive
functions and globals are; (2) metric budgets and the optimization goal.

For the first input, the user uses C attributes to make explicit what sensitive
functions/variables are. The following example shows how to specify \texttt{auth2} as a
sensitive function:

\begin{verbatim}
int (__attribute__((annotate("sensitive"))) auth2)
(char* userpwd, char* fn) { ... }
\end{verbatim}

The second piece of input is metric budgets and the optimization goal. Budgets
are in the form of \((b_c, b_f, b_s, b_x)\), where \(b_c\) is the budget for the sensitive-code
percentage, \(b_f\) the budget for the amount of sensitive information flow, \(b_s\) the
budget for the frequency of context switches, and \(b_x\) the pointer-complexity budget.
A user can allow some metrics have an unlimited budget, in which case \_ is used
for those metrics. Furthermore, the user specifies a metric as the target metric to
minimize; in notation, a “*” symbol is put after the budget to indicate that it is
the target metric. As an example, \((10\%, 2*, \_ ,\_)\) means that the budget for the
sensitive code percentage is 10\%, the budget for sensitive information flow is 2 bits,
budgets for the context-switch frequency and pointer complexity are unlimited, and
the goal is to minimize the amount of sensitive information flow.

With this input, PM then computes a partition in the form of a set of functions
and globals that should stay in the sensitive domain; the rest of the code stays in
the insensitive domain.

\footnote{We recently migrated the PDG-construction part to LLVM 9.0 and open sourced it
(https://bitbucket.org/psu_soslab/program-dependence-graph-in-llvm/src/master/).
Other parts of the tool are being migrated to LLVM 9.0 and will be released when mature.}

\textsuperscript{1}
To evaluate the quality of a partition and be able to compare different ways of partitioning for the same application, we overload the notation to also use a quadruple for the quality scores of a partition: \((c, f, s, x)\), where \(c\) is the sensitive-code percentage of the partition, \(f\) the amount of sensitive information flow, \(s\) the frequency of context switches, and \(x\) the pointer complexity. We added a feature to PM that takes a partition of a program as input and outputs its quality quadruple according to the program’s annotated PDG. This feature is useful when users want to use some initial partition’s quality scores as a starting point to find better partitions.

**LLVM passes.** Clang is used to compile the input program’s source code into LLVM IR code. The source code is assumed to include user annotations about where sensitive data is. PM adds LLVM passes at the IR level for PDG construction and performing measurements on code size and pointer complexity. Our PDG construction reuses PtrSplit, which allows modular PDG construction and relies on only local but not global pointer analysis. LLVM passes are also added to count the code sizes of functions and compute the pointer complexity for the types of functions and global variables. These measurements are then added to the PDG as weights.

**Measuring information flow.** PM measures sensitive information flow at the function level, in particular, during function calls and returns and between functions and globals. For this step, the input is a piece of sensitive information. The output is forward information flow \(fflow(e)\) and backward information flow \(bflow(e)\) for each edge in the input program’s PDG. For instance, if \(f_1\) calls \(f_2\) just once and passes a 32-bit secret password, the amount of forward flow is 32 bits; if \(f_2\) returns the comparison result between the password and a constant, the amount of backward flow is just 1 bit.

PM adapts Flowcheck [41] for measuring information flow; as far as we know, it is the only publicly available tool that produces quantitative information flow for realistic programs. It relies on dynamic analysis to track sensitive information flow at runtime for a particular input and uses a max-flow algorithm to quantify the amount of flow. Measuring information flow becomes feasible on a single run, with the downside that measurements may not apply to other runs. However, Flowcheck is designed to measure information flow between input and output at
the level of a whole program, while PM needs to measure information flow at
the function level. Appendix 3 presents in detail how PM adapts Flowcheck for
function-level measurement of information flow and how it aggregates information
flow over multiple runs. We next give a brief account.

Flowcheck is adapted to measure three kinds of information flow at the function
level: explicit, implicit, and potential flows. For explicit flows, when a function
gets called with some arguments, it measures how much sensitive information is
stored in the arguments and how much in the function’s return value. For implicit
flows, when a function contains a conditional jump that depends on sensitive
information, it tracks that one bit of flow and propagates it interprocedurally to
the function’s callers. Potential flows happen when pointers to sensitive buffers are
passed interprocedurally. Even if the callee function’s current code does not access
the underlying buffers, by our attack model an attacker may change the callee’s
computation (e.g., via return-oriented programming) to access those buffers, after
the callee has been taken over by the attacker. Therefore, in our context, it is
important to measure potential damage the attacker can cause by getting hold of
capabilities to access sensitive data and we call it potential flows. Note that this
kind of information flow is typically not considered in information-flow literature
since in that setting code is assumed to not change dynamically.

Measuring context-switch frequency. To determine the context-switch fre-
cquency when a call edge or a data-flow edge in the PDG becomes a boundary edge
after partitioning, PM uses Intel’s Pin tool [43] to profile program execution. During
the execution of a program, our Pin-based tool produces a logfile that records
caller-callee pairs of function calls, pairs of global variables and functions when
functions read from or write to global variables. Using the logfile, PM computes
the number of times a particular call site executes. If function $f_1$ can call $f_2$ at
multiple call sites, the call times for all call sites are summed into a total call time
from $f_1$ to $f_2$. Similarly, PM computes the number of times a global variable is read
or written by a function. Then, we divide the access amount with the execution
time to compute the frequency. Finally, average frequencies over multiple runs are
used as $af(e)$ in the PDG.

Integer programming solving. Given a PDG annotated with weights, PM
converts it into an integer-programming problem following Appendix 2. During
implementation, we discovered that currently popular RPC libraries (e.g., Sun RPC and Google’s gRPC) do not support bidirectional control transfers. An example is when function $f_1$ in domain 1 calls function $g$ in domain 2 and function $g$ in turn calls back function $f_2$ in domain 1 (e.g., via a function pointer). Due to this limitation, we add further constraints to our IP model so that only single-directional RPC is allowed.\footnote{ntirpc claims to have bidirectional RPC support, but their developers told us that the implementation was "sketchy" in private emails; when it becomes mature, we should be able to plug it into our tool and remove the single-directional RPC constraints from the IP model.} In detail, the new constraints allow only edges from the insensitive domain to the sensitive domain to appear on the RPC boundary; any function that the sensitive domain invokes (e.g., through callbacks) has to stay or replicated in the sensitive domain. The sensitive domain effectively becomes a passive server listening for RPC requests from the insensitive side. After conversion to IP, PM solves it using a generic integer programming solver. We use \texttt{lp\_solve}, although other integer programming solvers should also be compatible. The output of the solver tells us what functions and global variables should be in the sensitive domain and in the insensitive one.

\textbf{Implementing partitions.} PM is designed for producing information about how to partition a program. Implementing a partition still needs user involvement. It provides some automation for implementing a partition through process separation. In particular, it automatically generates interface definitions in Sun RPC’s IDL (Interface Definition Language). \texttt{rpcgen} is then run on the IDL code to generate interface code between the sensitive and the insensitive domain. The user then manually splits files, adjusts compilation scripts to link with the interface code, and compiles the original application into two executables: one for the sensitive domain, and one for the insensitive domain. During runtime, the two domains are loaded into separate OS processes and the process for the insensitive domain issues RPC calls to the process for the sensitive domain to request services.

In some circumstances, a partition can be implemented by other privilege-separation primitives such as dropping privileges or primitives based on hardware (e.g., Intel’s SGX). This will require further engineering to generate interface code that is compatible with specific privilege-separation primitives; we leave this for future work.
4.7 Evaluation

It can be hard for users to come up with the right budgets when using PM. By our experience, PM is best used as an interactive tool: a user starts with some initial budgets and gets a partition from PM. Based on that partition’s quality scores, the user adjusts the scores to make tradeoffs; the new set of budgets is then used by PM to produce a new partition, with its own quality scores. Multiple rounds may be needed before the user decides on the final partition. During the evaluation of PM, we used some high-level strategies to tune metric budgets in order to find good partitions, which are discussed next.

Strategies to tune budgets. The first kind of strategy is about how to get the initial budgets. An approach we found useful is to specify a target dimension for optimization, and apply unlimited budgets for all other dimensions. For instance, \((\ast, _, _, _)\) asks for the smallest sensitive domain, without constraining other dimensions. Another strategy is to use PM to get the quality scores of a known partition (such as the one that has only the sensitive functions/globals in the sensitive domain) and use those as the initial budgets.

The second kind of strategy is for making adjustments to budgets based on a particular set of partition scores to better satisfy the user’s goals. We discuss two such strategies:

- **Tradeoff strategy 1.** A user decreases the budget (i.e., improves the score) for a target dimension, sets an unlimited budget on a sacrifice dimension, and optimizes the sacrifice dimension. The intention is to produce a partition that trades off the sacrifice dimension for a bounded improvement on the target dimension, while making the least sacrifice on the sacrifice dimension. An example use of this strategy was for *thttpd* discussed later. We had a partition with quality \((9.15\%, 1.0, 1455.6, 9.0)\) and then we chose to trade off the sensitive-code percentage for a smaller context-switch frequency by specifying \((\ast, 1.0, 1455.5, 9.0)\). The new budgets led PM to find a partition with quality \((9.27\%, 1.0, 1411.2, 8.0)\); the performance improved due to a smaller context-switch frequency, which was obtained at the expense of a larger sensitive-code domain.

- **Tradeoff strategy 2.** A user reduces the budget on a target dimension, in-
creases the budget on a sacrifice dimension, and optimizes the target dimension. The intention is to produce a partition that trades off the sacrifice dimension for the best improvement on the target dimension within the budget for the sacrifice dimension. As an example, we had a partition with quality (9.15%, 1.0, 1455.6, 9.0) in `thttpd`. To follow strategy 2 to trade off the sensitive-code percentage for a smaller context-switch frequency, we used new budgets (10.00%, 1.0, 1455.5*, 9.0), which led to a new partition with quality (9.62%, 1.0, 1400.1, 8.0).

**Benchmarks.** We evaluated PM using a set of benchmarks listed in Table 4.1. The first four programs are small and from Linux’s shadow-utils package. For each benchmark, the table lists the name, the version, the source lines of code, the total number of functions, and the total number of globals. Further, it lists what sensitive data is used in our evaluation and the number of lines of annotations that are added to each program to mark sensitive data. Overall, the annotation burden is modest; most applications require only a few lines of annotations. For each benchmark, we designed an extensive set of test cases to collect security and performance metrics, as PM relies on some dynamic analysis for collecting measurements on the metrics we discussed.

With the collected measurements, we ran evaluation on each program to test whether PM can compute meaningful partitions with reasonable user guidance. For generated partitions by PM, we performed security and performance assessment. For those benchmarks that were also used in PtrSplit, we also compared PM’s results with PtrSplit’s results. All evaluation were on systems running x86-64 Ubuntu 14.04 with the Linux kernel version 3.19.0, an Intel Core i5-4590 at 3.3GHz,
4.7.1 Evaluation with thttpd

We evaluated PM on thttpd, an open-source http server program. The server is set up for receiving incoming connections and communications. Clients can connect to the server and, after authentication, request to download documents from a directory, called the top document directory, and its sub-directories set up by the server. thttpd stores user authentication information (username and password) in a file named .htpasswd. During authentication, a user provides a username and a password and thttpd looks up .htpasswd to check if there is a match. If there is, the authentication succeeds and follow-up actions requested by the user are authorized. Therefore, in this experiment, we treat the password file as sensitive data and perform partitioning to have a sensitive, high-secrecy domain that processes the password file.

In thttpd, two major functions are involved in authentication: auth_check (abbreviated as ac) and auth_check2 (abbreviated as ac2). If thttpd is configured to use a global password file, function ac first invokes ac2 and passes the user-input authentication data and the server top directory; ac2 then tries to open .htpasswd under the top directory and performs authentication. However, if .htpasswd is not found, ac2 returns failure to ac, which then calls ac2 again with the local directory from which the user requested a document. If a local password file is found, ac2 uses it to perform authentication. Therefore, for one user connection, ac may call ac2 twice.

4.7.2 Evaluation with wget

wget is a program for downloading files from web servers. We annotated the incoming data from a server as sensitive, since that data contained potentially malicious data (i.e., low-integrity data). The goal of partitioning is to produce a sensitive,
### Table 4.3. Partitioning choices for wget.

<table>
<thead>
<tr>
<th>Budgets ((b_c, b_f, b_s, b_x))</th>
<th>IP-Solving Time (s)</th>
<th>SCode(%)</th>
<th>Flow</th>
<th>CSwitch</th>
<th>Cplx</th>
<th>Overhead(%) (1M/1K)</th>
</tr>
</thead>
<tbody>
<tr>
<td>((_<em>, _</em>, _<em>, _</em>))</td>
<td>0.80</td>
<td>11.03</td>
<td>4047</td>
<td>1213.8</td>
<td>117.0</td>
<td>1493/6.2</td>
</tr>
<tr>
<td>((50%, 999.0%, 38.2, _*))</td>
<td>2.03</td>
<td>49.12</td>
<td>8</td>
<td>38.2</td>
<td>45.0</td>
<td>1.6/1.9</td>
</tr>
<tr>
<td>((16%, _<em>, _</em>, _*))</td>
<td>1.13</td>
<td>15.68</td>
<td>4052</td>
<td>198.5</td>
<td>137.0</td>
<td>412/7.2</td>
</tr>
<tr>
<td>((_<em>, 2.0, 38.2, _</em>))</td>
<td>1.56</td>
<td>78.42</td>
<td>2</td>
<td>38.2</td>
<td>14.0</td>
<td>1.5/2.3</td>
</tr>
</tbody>
</table>

This initial partition has a small sensitive domain, but the model reported high sensitive information flow and performance overhead; so this partition should not be adopted in practice. But to validate our performance model, we implemented the partition and collected its runtime overhead, shown in Table 4.3. The overhead was significant, consistent with the prediction of our performance model.

To get better performance and information-flow security, we set the budget on information flow dimension to be 999 and PM produced a more secure partition with quality \((39.81\%, 17.0, 1122.2, 74.0)\). This result implied that any partition for wget that prevents large sensitive information flow would contain a large sensitive domain. Therefore, we decided to relax the requirement on the sensitive-code percentage as a way of improving performance. By interactively using PM via similar strategies discussed earlier for thttpd, we got a partitioning choice that achieves a good balance between performance and security, with quality \((49.12\%, 8.0, 38.2, 45.0)\), which is shown as partition \(2\) in Table 4.3. The measured runtime performance overhead is much less than the first partition, with less than 2% overhead for the remote-server setting. This justifies the benefit of performing iterative refinement.

**Assessing security and performance.** We investigated the two partitions to understand why their security and performance were dramatically different. Fig. 4.4 presents the call graph of wget for the its main functions involved in implementing
the FTP protocol. In particular, main eventually invokes fd_read_body, which retrieves a file from an FTP server and writes the file content into a local file.

Partition 1 cuts between fd_read_body and lower-level functions; as a result, pointers to buffers holding the file content are passed from the sensitive domain to the insensitive one. This results in no protection from low-integrity data. Our model correctly predicts high information flow as it uses potential flow to measure the sizes of those buffers (which contain tainted data). Further, since fd_read_body is invoked many times (because it is transitively called by ftp_loop), this partition also results in bad performance due to many context switches. Partition 2, however, keeps ftp_loop in the sensitive domain, meaning that downloaded data does not cross the boundary, which achieves good integrity protection and a negligible overhead.

Based on this investigation, lifting the boundary to higher execution levels seems beneficial for reducing overhead and sensitive information flow, while moving the boundary to lower levels reduces the percentage of sensitive code. To validate this understanding, we used PM to discover partitions 3 and 4 in Table 4.3: the first cuts between getftp and fd_read_body, and the second cuts between main and retrieve_url. Partition 3 set the boundary at a lower level than 2; it reduced the percentage of sensitive code, at the cost of larger sensitive flow and performance overhead. For partition 4, the information flow was decreased. However, the majority of the program was in the sensitive partition.

Figure 4.4. Call graph and partitions for wget.
In all, partition 2 has the best balance between security and performance. We investigated its security in terms of how the sensitive domain can influence the insensitive one (for integrity protection). The reported 8 bits are all implicit information flows through return values, instead of more dangerous explicit and potential flows.

In terms of influence through the file system, the sensitive domain (1) writes the downloaded data to a local file and (2) writes data into a log file. None of these influences the sensitive domain, which does not read from those files.

Comparison with PtrSplit. We compared our best partition (2) with PtrSplit’s result on wget. Our partition achieves less runtime overhead than PtrSplit’s reported result (PM: 1.9% v.s. PtrSplit: 6.5%). However, our partition puts 304 functions into the sensitive domain, while PtrSplit reported only 8. After investigation, we realized that PtrSplit treated only the content of the downloaded file as sensitive, while we considered all data from the internet as sensitive. For example, communication messages between the server and wget are not treated as sensitive by PtrSplit and it puts functions that deal with such communication into the insensitive domain. Furthermore, PtrSplit did not count duplicated functions when reporting the size of the sensitive domain. When considering duplication, PtrSplit’s partition actually had 31.53% of code in the sensitive domain.

4.7.3 Evaluation with telnet

Telnet is a tool often used for controlling a remote machine. After a successful login, telnet sets up a bidirectional terminal-based communication interface. Data from the remote side is received and displayed in the local terminal; command-line operations are parsed from the local terminal and sent to the remote machine to be executed. Since telnet communicates with a remote server there is also a risk of receiving low-integrity data from the server. Our primary goal is to isolate the component that processes untrusted internet data.

Partitioning process. We first marked functions process_rings, netflush, and tn as sensitive during partitioning, since they interact with the internet. Then, we used a budget (\(\_\_\_\_\_\_*999_\_\_\_\_\_\_\)) to discover an initial partition, which is shown as partition 1 in Table 4.4. According to its quality, the smallest sensitive domain already contains a majority of the code. Therefore, we switched to search for a
low-overhead partition. In three iterations, we discovered partition 2 in Table 4.4.

**Assessing security and performance.** To understand why the sensitive domain had to be large, we investigated telnet’s source code and found that the main function in telnet directly invokes tn after parsing the command-line options. Since main has to stay in the insensitive domain and only single-directional RPC is supported, partition 2 can cut only between main and tn, which is near the top of execution. As a result, only main, the command-line parsing component, and functions that perform clean-ups (e.g., Exit) were put into the insensitive domain. Partitioning at other places would require bidirectional RPC support, as shown in Fig. 4.5.

Therefore, the fundamental reason was the lack of bidirectional RPC support when implementing partitions. To check whether allowing bidirectional RPCs would produce interesting partitions (although we would not be able to implement them), we configured PM to discover partitions with bidirectional boundaries. After several iterations, we discovered a partition with a small sensitive domain (13.10% code), shown as partition 3 in Table 4.4.

<p>| Table 4.4. Partitioning choices for telnet; “N/A” are for partitions that were not implemented. |
|-----------------------------------------------|-----------------|-----------------|---------------|-----------------|-----------------|-----------------|-----------------|-----------------|</p>
<table>
<thead>
<tr>
<th>Budgets ((b_c, b_f, b_s, b_x))</th>
<th>IP-Solving Time (s)</th>
<th>SCode(%)</th>
<th>Flow</th>
<th>CSwitch</th>
<th>Cplx</th>
<th>Overhead(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ((*, *999, _, _))</td>
<td>0.41</td>
<td>74.11</td>
<td>3.0</td>
<td>609.9</td>
<td>146.0</td>
<td>N/A</td>
</tr>
<tr>
<td>2 ((*, *999, 16.0, 15.0))</td>
<td>2.30</td>
<td>86.32</td>
<td>0.0</td>
<td>16.0</td>
<td>15.0</td>
<td>7.9</td>
</tr>
<tr>
<td>3 ((*, *999, _, _))</td>
<td>0.21</td>
<td>13.10</td>
<td>26.0</td>
<td>13305.0</td>
<td>227.0</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Assessing security/performance and comparison with PtrSplit.** Without the bidirectional RPC support, we cannot avoid a large sensitive domain for telnet. Through manual inspection we determined that the sensitive domain in 2 does not influence the insensitive domain through the file system (the sensitive domain writes only to stdout). In terms of runtime performance, our implementation had a lower runtime overhead compared with PtrSplit’s result (9.6%). After understanding PtrSplit’s result, we realized that partition 3 was the same as PtrSplit’s result. However, PM predicted large performance overhead for PtrSplit’s partition. After inspection, we believe that PtrSplit’s partition was manually rewritten to accommodate single-directional RPC and, as a result, not all functionality was preserved after partitioning.
4.7.4 Evaluation with nginx

.nginx is a web server and supports the username/password authentication. In our experiment, we partition nginx to protect the server-side password file from being leaked.

**Partitioning process and implementation.** For nginx, we marked function ngx_http_auth_basic_handler as sensitive, since it reads the password file. We started with the smallest sensitive domain that prevents potential flows resulting from pointers to the password file with budgets \((\_*, 999, \_, \_)\). As shown in Table 4.5, the metrics for partition ① indicate a large performance overhead. However, after inspecting the source code, we found there was only one function call across the boundary. Furthermore, all of the global variables used by the sensitive domain are read only. Therefore, we replicated those global variables and implemented data-synchronization by RPC. With such global variables duplicated, we have another boundary shown as ② in the table, which we implemented and collected the runtime overhead of partitioned authentication over unpartitioned authentication. Note that the overheads shown in the table are only for authentication; the partition does not incur overhead for common operations of nginx, such as serving web pages.

**Assessing security and performance.** In ②, the password file is only accessible
to the sensitive domain. The only one bit of leakage is the authentication response. As for the possible leakage through the file system, the sensitive domain may write to log files; however, the insensitive domain does not read from the log files. PtrSplit does not partition nginx; therefore, we did not compare with it.

### 4.7.5 Evaluation with Linux shadow-utils

We also experimented on a set of programs from the Linux shadow-utils package. There are over 30 small programs in this package. Many of them do not access security-sensitive information; for example, program "groups" just prints a user’s group information. Some of the programs are difficult to set up and experiment with; for example, "login" starts a login session. So we excluded those. For the remaining programs, we performed partitioning with PM. During the process, we realized that there were potential flows from the secret to the main functions in passwd and chage. However, since main had to stay in the insensitive partition, there would be no way of preventing the insensitive partition from holding sensitive data, for function-level partitioning. Hence, we manually changed the main functions of passwd and chage by extracting operations that read and update the password and shadow files to separate functions. The changed passwd and chage then became partitionable at the function level. The other two programs (chsh and useradd) required no changes.

For these four programs, we used PM iteratively to find one partition for each program and tested runtime overhead. We show the results in Table 4.6. Note that these programs are small, which excluded us from finding multiple interesting partitions.

### 4.7.6 Vulnerabilities mitigated by partitioning

The security metrics in PM are quantitative information flow and sensitive code percentage. There are many benefits of using these metrics. Another possible
Table 4.6. Partitioning choices for Linux shadow-utils

<table>
<thead>
<tr>
<th>Prog</th>
<th>SCode(%)</th>
<th>Flow</th>
<th>CSwitch</th>
<th>Cplx</th>
<th>Overhead(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>chsh</td>
<td>51.52</td>
<td>0.0</td>
<td>0.5</td>
<td>2.0</td>
<td>1.00</td>
</tr>
<tr>
<td>useradd</td>
<td>50.94</td>
<td>0.0</td>
<td>1781.4</td>
<td>29.0</td>
<td>11.33</td>
</tr>
<tr>
<td>passwd</td>
<td>82.33</td>
<td>0.0</td>
<td>846.2</td>
<td>13.0</td>
<td>7.50</td>
</tr>
<tr>
<td>chage</td>
<td>6.57</td>
<td>0.0</td>
<td>77.0</td>
<td>2.0</td>
<td>80.63</td>
</tr>
</tbody>
</table>

security metric is the amount of past known vulnerabilities that can be mitigated. We have argued against incorporating it into PM since it does not consider unknown vulnerabilities. On the other hand, if a partition can mitigate most of the past known vulnerabilities, it provides some evidence about the partition’s security strength. Therefore, we searched for all vulnerabilities in the National Vulnerability Database (https://nvd.nist.gov/vuln) for the versions of software we used in evaluation. We excluded those Linux shadow-utils programs as their vulnerability dataset is too small to draw any meaningful conclusion. Table 4.7 lists all CVEs for the versions of programs we used, and whether a CVE can be mitigated by a partition produced by PM.

Table 4.7. Mitigated vulnerabilities by different partitions.

<table>
<thead>
<tr>
<th>Program</th>
<th>Version</th>
<th>Vulnerability</th>
<th>Mitigated by</th>
</tr>
</thead>
<tbody>
<tr>
<td>thttpd</td>
<td>2.25</td>
<td>CVE-2013-0348</td>
<td>1 2 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CVE-2009-4491</td>
<td>1 2 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CVE-2006-4248</td>
<td>1 2 3</td>
</tr>
<tr>
<td>wget</td>
<td>1.18</td>
<td>CVE-2018-0494</td>
<td>2 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CVE-2017-6508</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CVE-2017-13090</td>
<td>2 3 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CVE-2017-13089</td>
<td>2 4</td>
</tr>
<tr>
<td>telnet</td>
<td>1.9.4</td>
<td>CVE-2005-0468</td>
<td>1 2 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CVE-2005-0469</td>
<td>1 2 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Exploit-DB-45982</td>
<td>multiple overflows (<a href="https://www.exploit-db.com/exploits/45982">https://www.exploit-db.com/exploits/45982</a>)</td>
</tr>
<tr>
<td>nginx</td>
<td>1.9.5</td>
<td>CVEs from 2016 to 2018</td>
<td>2</td>
</tr>
</tbody>
</table>

`thttpd` and `nginx` are about preserving confidentiality. According to our attack model, we consider a vulnerability mitigated by a partition if it resides in the insensitive (low-secrecy) domain of the partition. For any of the three partitions of `thttpd`, all CVEs we found can be mitigated since the CVEs reside in the insensitive domain.

---

3 multiple overflows https://www.exploit-db.com/exploits/45982
domain. Therefore, even if an attacker can successfully hijack the insensitive domain, she cannot steal passwords in the authentication file using the vulnerabilities. For *nginx* version 1.9.5, there are 8 CVEs in total. We inspected these CVEs and none of them resides in the authentication module, which means our partition 2 can mitigate all these vulnerabilities.

*wget* and *telnet* are about protecting integrity. According to our attack model, we consider a vulnerability mitigated by a partition if it resides in the sensitive (low-integrity) domain of the partition. Table 4.7 shows that the best partition PM found for *wget* (2) can mitigate three out of four CVE vulnerabilities; the best partition for *telnet* (2) can mitigate all three vulnerabilities.
Chapter 5  
Multi-module Privilege Separation

PtrSplit and Program-mandering successfully achieve quantitative privilege separation for programs with C-style pointers. However, both of them support only two-module program partitioning. This limitation means that during the whole separation process, only one kind of sensitive data can be tracked, quantified and finally isolated in one sensitive domain, which is far from enough for the protection needs of different users in reality. In this chapter, we introduce our work called Multi-module Partitioning (MMP), which supports partitioning a program into multiple modules to isolate different sensitive data in different domains separately.

5.1 Motivation

Figure 5.1 shows a sample program which is based on the well-known Oblivious Transfer (OT) Problem [46]: Alice has two secrets (alice_secret_1 and alice_secret_2) and Bob wants to learn alice_secret_i (i = 1 or 2). OT requires that: (1) Bob requests only one secret but does not want Alice to know his decision; (2) Alice releases to Bob only the secret that Bob requests but not both. It has been proved that in OT a trusted third party is needed if we want to meet the requirements of Alice and Bob at the same time (see [47]).

Accordingly, in Figure 5.1, there are two functions getAlice1 and getAlice2, getAlice1 opens a sensitive file called AliceSecret, and retrieves alice_secret_1; getAlice2 opens AliceSecret also but retrieves alice_secret_2. There is one
```c
int selectSecret (int n) { return n > 0; }
int transfer (int bob_secret) {
    int bob_secret = getBob();
    int alice_secret1 = getAlice1();
    int alice_secret2 = getAlice2();
    if (selectSecret(bob_secret) == 1)
        return alice_secret1;
    else
        return alice_secret2;
    return 0;
}
int getAlice1 () {
    FILE* fp = fopen("AliceSecret", "r");
    int secret1 = retrieveInt(fp);
    return secret1;
}
int getAlice2 () {
    FILE* fp = fopen("AliceSecret", "r");
    int secret2 = retrieveInt(fp);
    return secret2;
}
int getBob() {
    FILE* fp = fopen("BobSecret", "r");
    int secret = retrieveInt(fp);
    return secret;
}
```

Figure 5.1. A motivating example for MMP: oblivious transfer.

function called getBob that opens another sensitive file called BobSecret to retrieve Bob’s decision for request. Function transfer is an agent that simulates the oblivious transfer protocol. It takes a request (bob_secret) from Bob, and returns either alice_secret_1 or alice_secret_2 depending on bob_secret. In this toy program, transfer plays the role of the trusted third party. By calling transfer, we can successfully achieve the goal that OT targets for.

Although this toy OT program is short, it is enough to illustrate the necessity of multi-module partitioning for privilege separation. Clearly, there are two kinds of sensitive information in this OT program: Alice’s secrets and Bob’s secret. If we simply apply two-module program partitioning for privilege separation, the two kinds of exclusive sensitive information will be put into the same domain for
protection because two-module program partitioning only supports labeling and tracking a single kind of sensitive information. Hence, common program partitioning has to treat Alice’s secrets and Bob’s secret as a whole for processing, which fails to meet the requirements of both Alice and Bob.

Simply re-applying program-mandering has no guarantee on isolating different sensitive data separately, though we can annotate each kind of sensitive data and run PM for them separately. Figure 5.2 shows an intuitive case that PM fails to isolate sensitive data A and B in a monolithic program. In this case, A and B still stay together in the overlapping part of two sensitive modules, which means there must be some functions that can access A and B from one address space. The reason of this failure is that for our ultimate goal of "finding a satisfactory partition for sensitive data A and B", running PM for A without considering B can only find a local optimal partition for A, but not a global optimal partition for A and B. To find the latter, we need to redesign our partitioning algorithm to consider all kinds of sensitive data from a global perspective.

Figure 5.2. Partitioning a program with two kinds of sensitive data (use PM twice).
5.2 Multi-module Partitioning Algorithm

MMP’s workflow is quite similar to PM. MMP also takes the source code of an application with different annotations for each kind of sensitive data as input and constructs a PDG for it. Then, based on the PDG, MMP constructs a call graph and annotates its vertices and edges with weights that represent the security and performance measurements collected by the same approaches we used in PM. The main difference between MMP and PM is that MMP considers \( n \) \((n \geq 2)\) kinds of sensitive data. And accordingly, the output is not a sensitive partition and an insensitive partition any more, but \( n \) \((n \geq 2)\) sensitive partitions and an insensitive partition.

Figure 5.3 shows a two-module partitioning example for our OT program in Figure 5.1. We annotate two kinds of sensitive data, Alice’s secret and Bob’s secret, and annotate the PDG with collected measurements represented as weights on edges \((w_{13}, w_{23}, w_{34})\) and nodes \((c_1, c_2, c_3, c_4)\). Next, we calculate security/performance scores (represented as function \( f \)), and apply our multi-module partitioning algorithm according to the specifications. Finally, we partition the OT program into three modules: module 1 (\texttt{getAlice1} and \texttt{getAlice2}) interacts with Alice’s secret, module 2 (\texttt{getBob}) interacts with Bob’s secret, and module 3 (\texttt{transfer}) that works as an agent.

We formalize the basic definitions and metrics in multi-module partitioning
problem as follows:

**Definition 7** (Multi-module Partitions). There is a special function $s_i$ for reading sensitive data kind $i$ into the program. A multi-module partition of $G = (V, E)$, is modeled as $n + 1$ sets of functions $(S_0, S_1, S_2, ..., S_n)$: an insensitive domain $S_0 \subseteq V$ and $n$ sensitive domains $S_1, S_2, ..., S_n \subseteq V$ and they satisfy (1) $s_i \in S_i \ (i \in \mathbb{N}, \ 1 \leq i \leq n)$, and (2) $\bigcup_{0 \leq i \leq n} S_i = V$.

**Definition 8** (Optimal multi-module partitioning). For a set of metrics $M$, a target metric $m_k$ to minimize, and budgets $B = (b_1, b_2, ..., b_n)$, the optimal partition $P = (S_0, S_1, S_2, ..., S_n)$ is the one that minimizes the target metric and satisfies the following constraint: $\forall m_i \in M, m_i(P) \leq b_i$, where $m_i(P)$ is the value of metric $m_i$ for partition $P$.

In a partition $P = (S_0, S_1, S_2, ..., S_n)$, we use $\text{Flow}_j(S_u, S_v) (u \in \mathbb{N}, v \in \mathbb{N}, 0 \leq u \leq n, 0 \leq v \leq n)$ to represent the $j$-th sensitive data’s information flow capacity from $S_u$ to $S_v$. $\text{Flow}_j(S_u, S_u) = 0$ because we only consider the information flow between two different modules.

We use the total amount of information flow capacity to measure the $j$-th sensitive data from other modules to the $i$-th module (see Figure 5.4), which is:

**Definition 9** (Total sensitive information flow capacity).

$$C_{i,j} = \sum_{k=0}^{n} \text{Flow}_j(S_k, S_i) \ (i \neq j)$$

Next, we reconsider the four metrics for quantification in PM. We start with the information flow metric. For a program with $n$ kinds of sensitive data, by definition 7 we have $n + 1$ modules; so we need $n \times (n + 1)$ constraints to measure $n$ kinds of sensitive information flow among $n + 1$ modules.

Similar to PM, each vertex has a code size function noted as $sz(v)$. We use the total code size of all the sensitive domains in a partition $P = (S_0, S_1, S_2, ..., S_n)$ to redefine the sensitive code size metric, which is:

**Definition 10** (Total sensitive code size).

$$\text{SCode}(S_0, S_1, ..., S_n) = (\sum_{v \in V-S_0} sz(v))/(\sum_{v \in V} sz(v)).$$
We use the total amount of the context switch frequency among all modules to measure the frequency of all RPC calls on the boundary edges (BE) (remember af(e) stands for the access frequency of edge e):

**Definition 11** (Total context switch frequency).

\[
\text{CSwitch}(S_0, S_1, \ldots, S_n) = \sum_{e \in BE} \text{af}(e).
\]

Finally, we use the sum of the pointer complexity among all modules to measure the interface complexity of a partition in MMP:

**Definition 12** (Total pointer complexity).

\[
\text{Cplx}(S_0, S_1, \ldots, S_n) = \sum_{e \in BE} \text{plevel}(e).
\]

The metrics we proposed as above are not the only choice for measuring the quality of a partition in MMP. However, by using this design we can implement a simple user interface. For example, think about a program with \( n \) kinds of sensitive information, if we want to partition it into \( n + 1 \) modules with the most fine-grained design (pairwise design for each metric), the complexity of specification will be \( O(n \times n \times (n + 1)) = O(n^3) \), which is not only inefficient but also complicated for humans to specify.
Once we finish designing these metrics, we can formalize our specified constraints as a vector of form $<Scode, (C_{0,1}, C_{0,2}, \ldots, C_{i,j}, \ldots C_{n,n-1}), CSwitch, Cplx > (i \neq j)$. We reconfigured our IP model in PM to support processing $n$-dimensional ($n > 4$) vectors to do multi-module partition searching. Other technical modules such as the LLVM passes for computing pointer complexity can be reused without any change.

5.3 Discussion

We next briefly discuss and clarify some statements and situations in MMP. First, terms "partition" and "module" are used interchangeably in MMP. Since we use term "partition" in PM, we just want to be consistent in MMP. One thing we need to point out is, a module does not necessarily match to a cut in the call graph accurately. This is because we allow functions to be replicated to different modules. For some fundamental functions (e.g., logprintf) in a program, replicating them to different modules can greatly reduce the performance overhead cost.

Second, as to the secret data and its labels, commonly there is only one function that reads data from each kind of secret, and we call this function a special function. After the partitioning, a special function always stays together with its corresponding secret source. For example, in thttpd function auth_check2 reads secret data from the authentication file, so auth_check2 is a special function and it must stay together with the authentication file after the partitioning. In some more complex situations, there may be multiple functions that read data from the secret source. Accordingly, we need to use the same secret label to mark all the functions as special functions, which means we have a set of special functions instead of only one function for that secret source. However, we did not encounter these complex cases in our current experiments.

Third, the multi-module partitioning algorithm is almost the same as the algorithm in PM. For MMP, the real challenge is to design new metrics as inputs for the IP model and to implement inter-process communication among different modules. Once we have well-defined new metrics and finish profiling the call graph, we just need to change the parameter input for IP model from 2 to $n$ ($n \geq 3$) to support multi-module partitioning. The change of IP module configuration is trivial, and we are going to put this step in a README file in our future open-source code.
Last but not the least, it is worth to mention that by using MMP, we can separate for integrity and confidentiality in the same session. For example, in our
wget experiment we have both confidentiality and integrity stories at the same time. On one hand, the configuration file is protected from unauthorized access
and misuse (confidentiality); on the other hand, the module for receiving the
downloaded file (may contain malicious content) is also isolated so we protect the
left two modules from unauthorized alteration (integrity). By using MMP we can
support fine-grained privilege separation.

5.4 Evaluation

5.4.1 Evaluation with thttpd

We evaluated MMP on thttpd. We considered two kinds of sensitive data for thttpd:
(1) an authentication file that contains the user password; (2) a configuration file
that stores all the options for the server configuration. Since we are going to
partition thttpd into two sensitive modules and one insensitive module, and do not
hope two different sensitive information can flow into the module of each other, we
can simplify the vector of constraints on sensitive information flow by using only
two capacity values other than six ($2 \times 3 = 6$). Furthermore, for sensitive data
from the authentication file, it is a common sense that 1 bit of leakage is acceptable.
Therefore, we got an initial budget as (_*, (1, _), _, _) to start the first round
partition searching. In the first round, sensitive code percentage (SCode) is selected
to be optimized for our security priority. Based on the first round’s scores, we
want to pursue the best performance estimation. So, we switched the optimization
dimension to context-switch frequency with (1, 6) as the budget of the two kinds
of sensitive information flow; we got some improvement over context-switch as well
as interface complexity. However, the code percentage dimension behaves much
worse than the first round’s score. Naturally, we used the scores of second round
as the budget and rechose the code percentage dimension to optimize, wishing to
minimize the scarifce on security. Hence, we used budgets (_*, (1, 6), 2000.0, 26.0)
to find a new choice of scores (18.52, (1, 6), 2000.0, 26.0) (see (1) in Table 5.1).
Code in the sensitive partitions in total is still less than 20% of the original code.
Table 5.1. Three-module partitioning choices for \texttt{thttpd}: round 1 (RCode means replicated code. We list RCode specifically because we may have SCode $\geq$ 100% when SCode includes too much RCode).

<table>
<thead>
<tr>
<th>Budgets $(b_c, b_f, b_s, b_x)$</th>
<th>SCode (RCode)(%)</th>
<th>Flows</th>
<th>CSwitch</th>
<th>Cplx</th>
<th>IP-Solving Time (s)</th>
<th>Overhead(%)</th>
<th>1M/1K</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle $*,$(1,_),_\rangle$</td>
<td>13.43 (1.27)</td>
<td>(1.6)</td>
<td>2076.9</td>
<td>67.0</td>
<td>0.40</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>$\langle $_,(1.6),_\rangle$</td>
<td>64.03 (29.06)</td>
<td>(1.6)</td>
<td>2000.0</td>
<td>26.0</td>
<td>3.08</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>$\langle $__,(1.6),2000.0,26.0)$</td>
<td>18.52 (6.37)</td>
<td>(1.6)</td>
<td>2000.0</td>
<td>26.0</td>
<td>0.57</td>
<td>0.6/2.9</td>
<td>50.6/84.3</td>
</tr>
</tbody>
</table>

Table 5.2. Three-module partitioning choices for \texttt{thttpd}: round 2.

<table>
<thead>
<tr>
<th>Budgets $(b_c, b_f, b_s, b_x)$</th>
<th>SCode (RCode)(%)</th>
<th>Flows</th>
<th>CSwitch</th>
<th>Cplx</th>
<th>IP-Solving Time (s)</th>
<th>Overhead(%)</th>
<th>1M/1K</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\langle $*,$(1,_),_\rangle$</td>
<td>10.88 (0.85)</td>
<td>(1.6)</td>
<td>2076.9</td>
<td>67.0</td>
<td>0.40</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>$\langle $*,$(1.5),_\rangle$</td>
<td>16.62 (1.27)</td>
<td>(1.5)</td>
<td>3000</td>
<td>84</td>
<td>0.52</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>$\langle $_,(1.5),_\rangle$</td>
<td>63.74 (26.67)</td>
<td>(1.5)</td>
<td>2846.2</td>
<td>42.0</td>
<td>2.49</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>$\langle $__,(1.5),2846.2,42.0)$</td>
<td>22.78 (6.37)</td>
<td>(1.5)</td>
<td>2846.2</td>
<td>42.0</td>
<td>0.83</td>
<td>0.8/5.2</td>
<td>62.9/120.7</td>
</tr>
</tbody>
</table>

base; we believe this is a relatively good balance of all the dimensions. The results in round 1 are illustrated in Table 5.1.

We have not considered the second kind of information in the first round. Therefore, in the second round, we try to find the minimal information flow for the configuration data, while still maintaining 1 bit information flow for authentication data. So, the initial budgets are still $\langle $*,$(1,\_),\_\rangle$. The first score of the second kind of sensitive information flow is 6. We want to reduce it until we cannot. Therefore, we reduce one unit of this dimension from the current score to get a new budget 5. In all, we give budgets $\langle $*,$(1,5),\_\rangle$ and get a new choice of scores $(16.62, (1, 5), 3000, 84)$, which means 5 as the budget is feasible. Similarly with round 1, we then switch to balance between performance and security. We switch the optimization dimension to context-switch frequency. Then we use the new scores to maintain the improvement and minimize the security sacrifice. The whole process is listed in Table 5.2.

We implemented $\langle 1 \rangle$ and $\langle 2 \rangle$ to compare their runtime performance. Since they provide different security guarantees, it is hard to determine which is better. We would prefer the one with lower actual runtime overhead in our selection process. When downloading a 1Mb size file from a remote \texttt{thttpd} server, $\langle 1 \rangle$ and $\langle 2 \rangle$ have a tiny performance overhead difference (0.6% vs. 0.8%), which means both of them can be used to isolate the authentication file and the configuration file in \texttt{thttpd}.
Table 5.3. Three-module partitioning choices for wget.

<table>
<thead>
<tr>
<th>Budgets ((b_u, b_f, b_s, b_x))</th>
<th>SCode (RCode) (%)</th>
<th>Flows</th>
<th>CSwitch</th>
<th>Cplx</th>
<th>IP-Solving Overhead (%) (1M/1K)</th>
<th>Time (s)</th>
<th>Remote</th>
<th>Local</th>
</tr>
</thead>
<tbody>
<tr>
<td>((*, (999, 999), _, _))</td>
<td>45.04 (5.00)</td>
<td>(0,0)</td>
<td>18.0</td>
<td>801.0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>((_, (999, 999), *, _))</td>
<td>154.83 (59.29)</td>
<td>(0,0)</td>
<td>2.0</td>
<td>11.0</td>
<td>38.60</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>((_, (999, 999), 2.0, *))</td>
<td>143.57 (35.20)</td>
<td>(0,0)</td>
<td>2.0</td>
<td>9.0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>((*, (999, 999), 2.0, 9.0))</td>
<td>80.19 (17.49)</td>
<td>(0,0)</td>
<td>2.0</td>
<td>9.0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>((70.00, (999, 999), 2.0, *))</td>
<td>69.80 (24.80)</td>
<td>(0,0)</td>
<td>2.0</td>
<td>14.0</td>
<td>-</td>
<td>9.5/72.4</td>
<td>37.8/41.2</td>
<td></td>
</tr>
</tbody>
</table>

5.4.2 Evaluation for wget

For wget, we select two kinds of sensitive information: the received network data and the configuration file. Our major security goal is to avoid potential sensitive information flow. So, we got an initial budget as \((_, *, (999, 999), _, _)\) to start. Note that during the PDG annotation, “1000” was used to represent a potential flow; and the code-percentage was selected to optimize for the best security. The first round’s result is shown as \(\text{①}\) in 5.3. According to the result, we can see the code percentage dimension’s theoretical optimum is 45.04%, which is our baseline for exploring the alternatives.

Then, we switched to optimize the context-switch frequency and found the theoretical optimum as “2.0”. So, we used the score as a new budget for context-switch frequency to further search for the optimal choice for interface complexity, “9.0”. However, with performance optimized, the code percentage was unconstrained and became much worse than our baseline. Hence, we turned back to optimize the code percentage to be 80.19\% \((\text{①})\).

Compared to our baseline, there is still space for reducing the overhead of code percentage by sacrificing performance. Since context switch frequency is highly correlated to the runtime overhead, we did not want to sacrifice this dimension. We decided to sacrifice interface complexity score to improve the code percentage dimension. There are two ways to perform the tradeoff: (1) optimizing code percentage with an increased interface complexity budget; (2) optimizing interface complexity with a reduced code percentage budget. For the first approach, we decided to use the interface complexity score of the first intermediate choice to make the tradeoff; we used a budget, \((_, *, (999, 999), 2.0, 11.0)\), to yield \(\text{②}\). For the second approach, we selected “70.00” to further reduce the sensitive code and got an alternative \((\text{③})\). In all, we successfully traded interface complexity for a smaller sensitive partition.
Table 5.4. Three-module partitioning choices for telnet.

<table>
<thead>
<tr>
<th>Budgets ($b_l, b_f, b_s$)</th>
<th>SCode(RCode)(%)</th>
<th>Flows</th>
<th>CSwitch</th>
<th>Cplx</th>
<th>IP-Solving Overhead(%)</th>
<th>Time (s)</th>
<th>Remote</th>
<th>Local</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, 0)</td>
<td>27.24 (0.00)</td>
<td>(0, 0)</td>
<td>1904.0</td>
<td>700.0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(0, 0)</td>
<td>75.58 (12.38)</td>
<td>(0, 0)</td>
<td>20.0</td>
<td>95.0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(0, 0)</td>
<td>94.99 (21.28)</td>
<td>(0, 0)</td>
<td>20.0</td>
<td>67.0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(0, 0)</td>
<td>49.74 (0.18)</td>
<td>(0, 0)</td>
<td>20.0</td>
<td>67.0</td>
<td>-</td>
<td>0.6/2.9</td>
<td>50.6/84.3</td>
<td>-</td>
</tr>
<tr>
<td>(0, 0)</td>
<td>43.53 (0.19)</td>
<td>(0, 0)</td>
<td>20.0</td>
<td>95.0</td>
<td>6.37</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(0, 0)</td>
<td>40.00 (0.19)</td>
<td>(0, 0)</td>
<td>20.0</td>
<td>161.0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

5.4.3 Evaluation with telnet

Similar with wget, we also select two kinds of sensitive information: the received network data and the configuration file. For the same reasons, we start with budget (0, 0, (999, 999), _, _). We followed the exact strategy we applied for wget and found 6 alternatives (listed in 5.4). The initial one (0) shows a baseline for the whole exploring process. Then, switching optimization target to performance related dimensions gave us two intermediate choices to estimate the theoretical optimums for context-switch frequency and interface complexity. According to such two scores, we further yielded three alternatives (1, 2, and 3). 1 has the simplest RPC interface; 2 and 3 both sacrifice interface complexity for a smaller sensitive code partition, but in different ways. With a comprehensive consideration of all the three choices, we selected the simplest RPC interface to implement, which is choice 1.
Privilege separation is a technique in which software is partitioned into modules that are limited to the corresponding privileges they deserve to perform a specific task. Although privilege separation has been studied for many years, existing work still lack the ability to support C-style pointers, to balance between security and performance, and to isolate different kinds of sensitive data from a global perspective. This dissertation contributes a set of new techniques, PtrSplit, Program-mandering and Multi-module partitioning, to perform quantitative privilege separation with pointer supports. We believe these techniques can help programmers build systems more securely.

6.1 Summary of Our Current Work

Automatic prartitioning security-critical applications is an effective way of improving software security. We have proposed PtrSplit, a set of techniques that support general pointers in C applications, including parameter-tree-based PDG, selective pointer bounds tracking, and type-based marshalling/unmarshalling. These techniques push forward the state-of-the-art of privilege separating C applications and experiments suggest they have the potential of making automatic program partitioning practical.

We also proposed PM. PM is a quantitative framework for assisting privilege separation. It is based on our philosophy that, through quantitative information flow, a practical partition can be produced through a careful balancing between security and performance. This balancing cannot be fully automated as it has to take user requirements into account. PM provides users an interactive way for
exploring partitioning choices, while making their intentions explicit via budgets and a goal. Our experiments with real applications suggest that PM, while with some limitations, lets users explore the partitioning space in a principled fashion, helps users produce partitions that would be hard to obtain manually, and finds partitions to balance security and performance better.

MMP is a more general framework for fine-grained privilege separation. MMP treats different kinds of sensitive information separately, and isolates each sensitive data in a specific module. MMP achieves quantitative privilege separation for multiple kinds of sensitive data, and finds optimal partitioning boundaries among multiple modules from a global perspective. Out experiments show that MMP works well on practical software.

6.2 Future Work

In this section, we discuss our opinions and perspectives on our future work.

First, PtrSplit can be further optimized by using more efficient and robust pointer bounds tracking tools to reduce the overhead and enhance the availability. Recent work such as Low-Fat Pointers [48,49], CUP [50] and Checked C [51] provide general pointer tracking with lower overhead than SoftBound. We wish to combine our SPBT approach and these state-of-the-art bounds tracking techniques to make PtrSplit more efficient.

Second, we wish to replace Sun RPC with more efficient and modern frameworks such as gRPC [37]. We used Sun RPC in our current experiments because all the benchmarks are well-known software written in C, and Sun RPC supports C robustly and efficiently. However, if we are going to extend our framework to support more software written in other high level languages (even including C++), keeping using Sun RPC may become a serious limitation.

Third, our optimization frameworks in PM and MMP currently support only one optimization metric. A natural alternative to having four metrics would be to have a weighted sum of the four metrics so that we can optimize a single linear function of all four metrics in one step. We will need to further study methods to produce weights for this alternative and the effectiveness of those methods.

Fourth, PM and MMP’s implementations rely on dynamic analysis for measuring information flow and context-switch frequency. On the one hand, dynamic analysis
is the only known technique for measuring information flow rates in realistic programs. Most past studies on using static analysis to measure information flow (see [52] for a recent survey) have been theoretical and not produced practical tools. For instance, Clark et al. [53] described a static analysis that overapproximates quantitative information flow in programs. However, it is on an idealized language that does not support function calls, memory allocation/deallocation, and many other features. On the other hand, dynamic analysis applies to particular runs and requires a set of test cases. Designing test cases with good coverage is difficult; this issue can be mitigated to a certain degree by deriving test cases based on typical use cases and techniques such as fuzzing.

Fifth, by generating the optimal partition, PM and MMP automatically compute where data should be declassified. This enables automatic computation of declassification points for patterns such as authentication, which compress sensitive information. However, it does not work well for declassification patterns that scramble sensitive information. A typical example is encryption, in which dynamic information-flow tracking would report the amount of sensitive information flow from the key to the ciphertext is the key size. For these cases, additional techniques or manual declassification would be needed.

Finally, similar to other tools, our systems perform partitioning at the level of functions. As discussed before, partitioning a program at a granularity finer than functions, such as basic blocks or instructions, is sometimes necessary to produce good partitions. This issue is exacerbated by the lack of bidirectional RPC support, as demonstrated by telnet and some of the shadow-utils programs. When a top-level function \( f \) in the call graph (e.g., \texttt{main}) accesses sensitive data, all functions \( f \) invokes transitively have to stay in the sensitive partition, implying a large sensitive domain. This issue can be resolved by either providing bidirection RPC or splitting \( f \) (as demonstrated by shadow-utils programs). Implementing finer-grained partitioning would pose no theoretical difficulty, but introduce engineering and practical challenges in terms of collecting measurements at a finer granularity and implementing partitions.
Appendix | Critical Background Techniques in PM

1 Program-dependence-graph partitioning

To model a program with both functions and global variables, we use a Program Dependence Graph (PDG). In the PDG, vertices represent either functions or globals. We write $FV$ for the set of functions, $GV$ for the set of globals. We have $V = FV \cup GV$.

Edges represent either call edges or data-flow edges. Data-flow edges have two kinds: read edges and write edges. If function $f$ reads a global $g$, there is a directed read edge from $g$ to $f$. On the other hand, if function $f$ writes to a global $g$, we add a directed write edge from $f$ to $g$. We write $CE = \{e_{ij} \mid i, j \in FV\}$ for the set of call edges, $RE = \{e_{ij} \mid i \in GV \land j \in FV\}$ for the set of read edges, and $WE = \{e_{ij} \mid i \in FV \land j \in GV\}$ for the set of write edges. We have $E = CE \cup RE \cup WE$.

In PDG partitioning, we further allow globals to be sensitive. A partition $P = (S, T)$ is defined as before, except that $S$ and $T$ are now sets of functions and globals. $R = S \cap T$ is the set of duplicated functions and globals.

There are three kinds of forward boundary edges: (1) forward boundary call edges $FB_C = \{e_{ij} \in CE \mid i \in S \land j \in T - R\}$; (2) forward boundary read edges $FB_R = \{e_{ij} \in RE \mid i \in S - R \land j \in T\}$; (3) forward boundary write-edges $FB_W = \{e_{ij} \in WE \mid i \in S \land j \in T - R\}$. We have $FB = FB_C \cup FB_R \cup FB_W$. Similarly, there are three kinds of backward boundary edges: (1) backward boundary
call edges $BB_C = \{e_{ij} \in CE \mid i \in T \land j \in S - R\}$; (2) backward boundary read edges $BB_R = \{e_{ij} \in RE \mid i \in T - R \land j \in S\}$; (3) backward boundary write edges $BB_W = \{e_{ij} \in WE \mid i \in T \land j \in S - R\}$. We have $BB = BB_C \cup BB_R \cup BB_W$.

Furthermore, weights presented in Sec. 4.5 are also adjusted. First, a node for a global variable has zero code size. Second, information-flow weights are added for data-flow edges. Information can flow only along the direction of edges; that is, information flows to a function from a global variable on a read edge and flows to a global variable from a function on a write edge. Therefore, conceptually there should be no backward flow on data-flow edges. But to be uniform with the weights on call edges, we still give two weight functions for a data-flow edge $e$: $f\text{flow}(e)$ is the amount of information flow on the edge and $b\text{flow}(e)$ is always zero.

Third, for a read edge $e$ from global $g$ to function $f$, we use $af(e)$ for the frequency of $f$ reading from $g$ during profiling; similarly, for a write edge $e$ from function $f$ to global $g$, $af(e)$ is the frequency of $f$ writing to $g$ during profiling. Lastly, for a data-flow edge $e$ that connects a function $f$ to a global $g$, $p\text{level}(e)$ represents the complexity of $g$’s type signature. This is because, if the function and the global are in separate domains, the function has to read/write the global through an RPC to a getter or a setter function. Therefore, the type complexity of the global is used to represent the complexity of implementing the RPC.

With these adjustments, the definitions of sensitive code percentage, sensitive information flow, context-switch overhead, pointer complexity, and optimal partitioning are exactly the same as the case for call graphs and are not repeated.

### 2 Encoding optimal partitioning as IP

**Solution variables and objective.** We first declare two binary variables $\alpha_v$ and $\beta_v$ for each vertex $v$ in the PDG. Recall that a vertex represents either a function or a global variable. $\alpha_v$ is 1 iff $v$ is in the sensitive partition $S$ but not replicated; $\beta_v$ is 1 iff $v$ is in the insensitive partition $T$ but not replicated. That is, they satisfy $v \in S - R \iff \alpha_v = 1$ and $v \in T - R \iff \beta_v = 1$. As a result, $v \in R$ (v is replicated) iff $\alpha_v = 0 \land \beta_v = 0$. We term the two kinds of variables as solution variables.

For the objective function, we use the goal of minimizing sensitive code percentage as an example; other objective functions can be modeled in a similar way. Since
the total code size of the input program is a constant, minimizing the sensitive
code percentage can be converted to minimizing the code size in $S$, which is the
same as maximizing the code size in $T - R$, where $R = S \cap T$. Therefore, we can
use the following objective function:

$$
\max \sum_{i \in V} sz(i) \cdot \beta_i.
$$

**Intermediate variables and constraints.** The following constraints model
that (1) the special sensitive function or the special sensitive global variable $s$
must be in $S - R$ only, and (2) every function or global variable $i$ cannot stay in both
$S - R$ and $T - R$:

$$
\alpha_s = 1 \land \beta_s = 0 \land \forall i, \alpha_i + \beta_i \leq 1.
$$

When $\alpha_i + \beta_i = 0$, it means that the function or global variable $i$ represents is
replicated (that is, it is in $R$).

Since the direction of an edge matters when measuring sensitive information
flow, we further declare two intermediate variables $x_{ij}$ and $y_{ij}$ to represent if
the edge is a forward boundary edge or a backward boundary edge. Specifically,
$x_{ij} = 1 \iff e_{ij} \in FB$; and $y_{ij} = 1 \iff e_{ij} \in BB$.

With the input budgets $b_c, b_f, b_s,$ and $b_x$, we can construct the following con-
straints for all measurements based on Def 2:

$$
\left( \sum_{i \in V} sz(i) \cdot (1 - \beta_i) \right) / \text{totalSize} \leq b_c;
$$

$$
\sum_{i,j \in V} fflow(e_{ij}) \cdot x_{ij} + bflow(e_{ij}) \cdot y_{ij} \leq b_f;
$$

$$
\sum_{i,j \in V} af(e_{ij}) \cdot (x_{ij} + y_{ij}) \leq b_s;
$$

$$
\sum_{i,j \in V} plevel(e_{ij}) \cdot (x_{ij} + y_{ij}) \leq b_x.
$$

The first constraint limits the sensitive code percentage, assuming totalSize is the
total code size. The second limits the total of sensitive information flow. The third
limits the RPC context-switch frequency during runtime. And the fourth limits the
pointer complexity.
The next step is to constrain variables \( x_{ij} \) and \( y_{ij} \) with their related four solution variables \( \alpha_i, \alpha_j, \beta_i, \) and \( \beta_j. \) In our problem formalization, we have three different boundary edge sets for three types of edges. Therefore, constraints are introduced differently for different types of edges. We first discuss what logical formulas need to be encoded for each type of edges and then present how those logical formulas can be encoded by IP inequality constraints. For an edge \( e_{ij} \) from vertex \( i \) to vertex \( j \) in the graph,

1) if \( e_{ij} \) is a call edge,

\[
x_{ij} = 1 \iff e_{ij} \in FB_C \iff \beta_i = 0 \land \beta_j = 1 \iff \neg \beta_i \land \beta_j;
\]
\[
y_{ij} = 1 \iff e_{ij} \in BB_C \iff \alpha_i = 0 \land \alpha_j = 1 \iff \neg \alpha_i \land \alpha_j;
\]

2) if \( e_{ij} \) is a read edge,

\[
x_{ij} = 1 \iff e_{ij} \in FB_R \iff \alpha_i = 1 \land \alpha_j = 0 \iff \alpha_i \land \neg \alpha_j;
\]
\[
y_{ij} = 1 \iff e_{ij} \in BB_R \iff \beta_i = 1 \land \beta_j = 0 \iff \beta_i \land \neg \beta_j;
\]

3) if \( e_{ij} \) is a write edge,

\[
x_{ij} = 1 \iff e_{ij} \in FB_W \iff \beta_i = 0 \land \beta_j = 1 \iff \neg \beta_i \land \beta_j;
\]
\[
y_{ij} = 1 \iff e_{ij} \in BB_W \iff \alpha_i = 0 \land \alpha_j = 1 \iff \neg \alpha_i \land \alpha_j.
\]

To transform the above logical formulas into linear inequations, we use two classic IP techniques: (1) \( \neg x \) is equivalent to \( 1 - x; \) and (2) the relation \( y = 1 \iff x_1 \land x_2 \land \cdots \land x_n \) can be linearly modeled as

\[
y \leq x_i, \quad \forall i = 1, 2, \ldots, n
\]
\[
y \geq x_1 + x_2 + \cdots + x_n - (n - 1).
\]

For brevity, we only show how \( x_{ij} \) and \( y_{ij} \) are constrained when \( e_{ij} \) is a call-edge:

\[
x_{ij} \leq 1 - \beta_i,
\]
\[
x_{ij} \leq \beta_j,
\]
\[
x_{ij} \geq \beta_j - \beta_i,
\]
\[ y_{ij} \leq 1 - \alpha_i, \]
\[ y_{ij} \leq \alpha_j, \]
\[ y_{ij} \geq \alpha_j - \alpha_i. \]

So far, we have declared \(2|V| + 2|E|\) binary variables and constructed \(|V| + 6|E| + 5\) constraints.

### 3 Measuring information flow

In Flowcheck users specify what file opened by the program or what buffer used by the program is sensitive. Flowcheck’s dynamic analysis then constructs a flow graph during program execution. For a relevant operation during execution, a graph structure is generated to represent the sensitive information flow happened in the operation. Edges in the graph represent how sensitive data is processed in the program and are annotated with the amount of sensitive data being processed; that is, edges represent explicit information flows. For instance, a comparison between a 32-bit secret with a constant would produce (1) a 32-bit edge from the node for the secret to a new node for the comparison, and (2) a 1-bit edge from the comparison node to a new node for the comparison result. Implicit flows are also reported at the instruction level. If Flowcheck encounters a conditional jump and the processor flag that the jump depends on has 1-bit sensitive information (because of an earlier instruction that sets the flag using sensitive information), then Flowcheck reports that the jump has one bit of implicit flow.

The flow graph constructed by Flowcheck, however, does not directly report inter-procedural information flow PM is interested in. Next we discuss how this is calculated in PM on top of information provided by Flowcheck. This is presented in several steps: we first discuss how explicit flows through arguments, return values, and global variables are quantified and an optimization method for improving precision; we then discuss how implicit flows are treated; finally, we briefly discuss how PM aggregates flow quantities across multiple calls and multiple runs.

**Explicit flows.** When a function gets called with some arguments, PM needs to know how much sensitive information is stored in the arguments and how much in the function’s return value. The flow graph constructed by Flowcheck, however,
does not directly give such information, as explained below.

First, Flowcheck generates a graph structure for an operation only when sensitive information is involved in the operation. Function calls/returns, on the other hand, do not directly manipulate sensitive information. Take the following code as an example. For clarity, this example and other examples use a pseudo-code syntax, instead of the x86 assembly code syntax; in particular, we use “:=” for an assignment.

\[
eax := \text{ebx} \text{ xor } \text{ecx}
\]

\[
... 
\]

\[
\text{ret}
\]

In the default x86 calling convention, register eax contains the return value at the end of the function. Thus PM needs to know how much sensitive information is in eax when ret is executed. However, since ret itself does not manipulate eax, Flowcheck does not generate a graph structure related to eax. It instead would generate a graph structure when eax was assigned earlier in “eax := ebx xor ecx”, assuming ebx or ecx contains sensitive information. Consequently, PM would have to trace back from ret to the earlier assignment and use the assignment’s graph structure to know the amount of information in eax at the time of the return.

Second, Flowcheck uses an optimization to avoid generating a huge flow graph; it generates graph structures for operations that combine different pieces of data or transform data, but not when data is moved around completely unchanged. Take the following as an example:

\[
edx := \text{ebx} \text{ xor } \text{ecx}
\]

\[
... 
\]

\[
eax := edx
\]

\[
... 
\]

\[
\text{ret}
\]

A graph structure is generated for “edx := ebx xor ecx”, assuming ebx or ecx contains sensitive information; however, no graph structure is generated for “eax := edx” since it only moves sensitive information around without changing it. This example shows that, to calculate the amount of sensitive information in eax at the place of a return, one could perform dependence analysis to identify the last operation that affected eax and for which some graph structure was generated.
PM adopts an easier solution, which performs assembly-level rewriting to force Flowcheck to generate graph structures for function arguments and return values at the places of function calls and returns. Source code is first compiled to assembly code by using the x86 cdecl calling convention. At the assembly code level, a sequence of “eax := not eax; eax := not eax” is inserted before a return. This sequence was chosen because (1) the net effect of the sequence is a no-op: no registers or flags are affected;\(^1\) (2) if eax contains sensitive information, the not operations force Flowcheck to generate graph structures immediately before the return instruction, making it easy for PM to identify the amount of sensitive information in eax at the time of the return.

Similar rewriting is performed for function arguments and global variables so that PM can identify the amount of sensitive information in arguments and global variables during runtime. For function arguments, in the default x86 calling convention, arguments are passed on the stack. Before a function call, move instructions are used to move arguments from registers to the stack. Therefore, before such a move instruction, a sequence of “r := not r; r := not r” is inserted, assuming r is the register used in the move. Reads from or writes to global variables are also realized through move instructions. These move instructions are identified with the help of symbol tables, which tell where global variables are stored and a similar sequence of “r := not r; r := not r” is inserted before such a move.

We note that the rewriting is performed purely for measuring sensitive information flow. After the measurement, the rewritten program is discarded and PM’s partitioning is performed on the original program.

**Mincut for better precision.** After getting the amount of sensitive information in function arguments, return values, and global variables, one could directly add those numbers as weights to the PDG. For instance, if at a function call there are two arguments and each is measured to have 32-bit sensitive information, we could say that there are 64 bits of flow for the function call. However, the problem is that the two arguments’ information may overlap and the actual amount of sensitive information may be less than 64 bits.

To improve precision, PM performs a refinement. We discuss the case for function arguments; the cases for return values and global variables are similar.

---

\(^1\)The “not” instruction in x86 is like C’s one’s complement (~) operation, but not C’s logical not (!) operation.
For a function call’s arguments, the refinement (1) starts from the nodes for the arguments, (2) performs backward reachability on the flow graph to find a subgraph of nodes that can reach the starting nodes, up to $k$ nodes, and (3) then performs the mincut algorithm on the subgraph to find the max capacity of sensitive information in the starting nodes.

As a toy example, suppose there is a 32-bit secret, and a function call passes two arguments; the first argument is a copy of the secret, and the second is the result of one’s complement of the secret. The following figure shows the relevant graph structure generated by Flowcheck for this example.

The mincut algorithm tells us that the amount of information in the two arguments is just 32 bits, since both are derived from the same 32-bit secret. In our implementation, the threshold $k$ for the subgraph size is 10. We note that any $k$ would affect only precision, not soundness.

**Implicit flows.** When executing a conditional jump instruction that depends on sensitive information, Flowcheck would report that there is an implicit flow. However, the implicit flow is not propagated further by Flowcheck. For instance, if there is a subsequent operation that assigns a constant to eax, no graph structure is generated for the assignment even though eax contains sensitive information because the assignment is dependent upon the conditional jump.

To alleviate this, PM propagates implicit flows interprocedurally and aggregate them with explicit flows. As an example, suppose $f_1$ calls $f_2$ and there is a 1-bit implicit flow in $f_2$ because it contains a secret-dependent conditional jump; and its
return value contains 2-bit secret information because of explicit flows. Then in the PDG constructed by PM, the backward flow for the edge from \( f_1 \) to \( f_2 \) (i.e., \( bflow(e) \)) is annotated with 3 bits (by adding the quantities of implicit and explicit flows). Furthermore, the 1-bit implicit flow from \( f_2 \) to \( f_1 \) is propagated in the PDG following both data dependence and control dependence. For instance, if the call from \( f_1 \) to \( f_2 \) is caused by a call from \( h \) to \( f_1 \) and \( f_2 \)’s return value has 1-bit implicit flow, then \( f_1 \)’s return value is also considered to have a 1-bit implicit flow when it returns to \( h \).

**Potential flows.** In an unpartitioned program, passing a pointer that points to a secret between functions does not necessarily cause the secret information to flow into the callee function, because the pointer itself is not sensitive. For example, suppose function \( f \) calls \( g \) with a pointer that points to the secret, and \( g \) passes the pointer to \( h \) but does not deference the pointer. Then there is no explicit information flow in \( g \) reported by Flowcheck since no manipulation of secret information is performed in \( g \).

However, after partitioning, a function call is turned into an RPC, during which PM performs deep copying on pointers. For the same example above, if \( f \) and \( g \) are in separate partitions, the call from \( f \) to \( g \) is turned into an RPC, whose deep copying not only copies the pointer but also the secret data the pointer points to. As a result, the partition where \( g \) resides has the potential of reading the secret data through the pointer, if the partition is taken over by an attacker. In other words, even if \( g \) itself does not perform dereferencing, if the partition where \( g \) is taken over, the attacker may have the ability of inducing arbitrary computation within \( g \)’s partition and get the secret. To measure the potential information flow, PM marks the pointer that points to sensitive data and performs static tainting to locate function invocations that pass tainted pointers (i.e., pointers to sensitive data). For example, if \( f_1 \) calls \( f_2 \) with a pointer to a secret encryption key of size 1K, then the amount of potential flow is 1K, since \( f_2 \) has the potential of dereferencing the pointer to get the secret key.

**Aggregation over multiple calls and runs.** Flowcheck is a dynamic analysis tool; therefore, during the execution of a program, a function \( f_1 \) may call \( f_2 \) multiple times. The steps discussed so far produce a flow quantity for each call and a flow quantity for each return. Since PM produces a static PDG in which there is only
one edge from $f_1$ to $f_2$, it aggregates flow quantities associated with multiple calls. In particular, forward information flow $f\text{flow}(e)$ is the sum of forward flow quantities in multiple dynamic calls that correspond to the same call edge $e$; the same goes for $b\text{flow}(e)$. A similar aggregation process happens when a function reads from or writes to some global variable multiple times.

Dynamic analysis also suffers from the problem of code coverage. To alleviate the issue, in experiments we designed an extensive suite of test cases for each benchmark and ran the benchmark multiple times with different tests; PM then aggregates the flow quantities over multiple runs. In particular, for a call edge (or a data-flow edge), PM takes the max quantity over multiple runs. The hypothesis is that there is a single number that represents the maximum amount of information a single run of the program could ever produce; then the maximum from the individual tests is the best under-approximation of that ideal measurement. Another way of aggregation is to add flow amounts over multiple runs and is a conservative way of counting the amount of information flow through the whole test suite.
Bibliography


Vita
Shen Liu

Education

- **Ph.D. candidate in Computer Science** Jan. 2016 ~ May 2020
  The Pennsylvania State University, State College, PA
- **M.S. in Computer Science** Sept. 2008 ~ Mar. 2011
  Zhejiang University, Hangzhou, China
- **B.S. in Software Engineering** Sept. 2004 ~ June 2008
  Zhejiang University, Hangzhou, China

Professional Experience

*Research Assistant*, **Penn State**, University Park, PA Jan. 2016 ~ Present
- Worked on PtrSplit, Program-mandering and Multi-module partitioning projects for quantitative privilege separation with pointer supports.

*Research Intern*, **Microsoft Research**, Redmond, WA May 2018 ~ Aug. 2018
- Implemented bounds declarations checking and bounds widening for null-terminated arrays with C++/LLVM/Clang on *Checked C*, an open source project that makes C safer by adding bounds-checking to C pointers.

- Worked on formal methods research and Program Depdence Graph construction in LLVM.

*Software Engineer*, **Tencent Inc.**, Beijing, China July 2011 ~ Apr. 2012
- Developed server applications for Tencent Microblog with C/C++/MySQL on online tweet ranking, friend recommendation and log file system.

Publications during the Ph.D. study