Program-mandering: Quantitative Privilege Separation

Shen Liu*
Dongrui Zeng*
sxl463@cse.psu.edu
dxz16@cse.psu.edu
The Pennsylvania State University
University Park, Pennsylvania

Yongzhe Huang
yzh89@psu.edu
The Pennsylvania State University
University Park, Pennsylvania

Frank Capobianco
fnc110@psu.edu
The Pennsylvania State University
University Park, Pennsylvania

Stephen McCamant
mccamant@cs.umn.edu
University of Minnesota
Twin Cities, Minnesota

Trent Jaeger
tjaeger@cse.psu.edu
The Pennsylvania State University
University Park, Pennsylvania

Gang Tan
gtan@psu.edu
The Pennsylvania State University
University Park, Pennsylvania

ABSTRACT

Privilege separation is an effective technique to improve software security. However, past partitioning systems do not allow programmers to make quantitative tradeoffs between security and performance. In this paper, we describe our toolchain called 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 a previous tool that requires manual declassification.

CCS CONCEPTS

• **Security and privacy** → **Software and application security**; • **Software and its engineering** → *Automated static analysis*; *Dynamic analysis*.

KEYWORDS

Automatic program partitioning, privilege separation, integer programming

*Both authors contributed equally to this research and are co-first authors.

CCS '19, November 11–15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3354218

ACM Reference Format:
Shen Liu, Dongrui Zeng, Yongzhe Huang, Frank Capobianco, Stephen McCamant, Trent Jaeger, and Gang Tan. 2019. Program-mandering: Quantitative Privilege Separation. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3319535.3354218

1 INTRODUCTION

Privilege separation in software systems refers to the process of decomposing a system into multiple modules, each loaded into a separate protection domain. Privilege separation prevents a software system from being compromised completely by a single vulnerability because any compromised protection domain cannot directly access the code or data of the parts of the system running in other protection domains. Calls to functions in other protection domains are converted into remote procedure calls (RPCs) and data access is restricted to protection domains where necessary.

While there is clear potential to improve software security through the use of privilege separation, programmers face challenges in leveraging privilege separation to achieve security guarantees, to refactor software systems into working modules, and to maintain efficient performance in the decomposed system. First, systems that are privilege separated often aim to assign *least privilege* [22] permissions to each protection domain, but it is unclear whether such permission assignments achieve security guarantees. For example, OpenSSH was manually refactored by Provos et al. [20] into one privileged server process and many unprivileged monitor processes, each of which handles a user connection. In this setup, a compromised monitor process should not affect the server process or other monitor processes. However, only later was it shown that the privilege separated OpenSSH achieved an approximation of the strong Clark-Wilson integrity model [23].

Second, manually privilege separating complex software is labor intensive. Automatic privilege separation aims to partition software with little user involvement. For instance, Privtrans [4] takes as input C source code and user annotations about sensitive data and declassification, and employs static analysis to separate the application into a master process that handles privileged operations and an unprivileged slave process. Automatic privilege separation has great potential for improving software security.

Third, 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.

Therefore, being able to achieve security goals while still allowing users to make tradeoffs between security and performance is essential for the success of privilege separation. We call security and performance *partitioning factors*, as they are critical for partitioning. Many previous tools [4, 14–16, 21, 29, 30], however, consider only security during privilege separation. Some of these tools (e.g., [4, 15]) require user-specified data declassification, a process that allows sensitive data flow from a *sensitive domain* (i.e., the domain that processes sensitive data) to an *insensitive domain*, to prevent leakage of secret data and/or use of low-integrity data. For example, suppose there is an authentication function that uses a secret password and a client function f that invokes the authentication function. By declassifying the return value of the authentication function, f can then be put into the insensitive domain. This manual declassification process, however, burdens programmers as they have to decide where to perform declassification. In general, programmers have to evaluate the performance impact of boundary crossings and the possible security impact of information flows at those boundaries, as well as the requirements for writing effective declassifiers at each candidate boundary.

A few past systems [6, 12, 28] consider both security and performance during partitioning, but none of them quantifies security and supports users in making quantitative tradeoffs between security and performance. We propose a new automatic privilege-separation framework, called Program-mandering[1], abbreviated as PM. It makes a key observation that many applications' security goals are related to information flow and it therefore adopts quantitative information flow as a metric for security. Consequently, 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 quantitative 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.

2 RELATED WORK

Several tools have been proposed to assist programmers in manually partitioning their applications, including Privman [13] and Wedge [3]. However, they require programmers to manually figure out a good partition boundary. A number of tools [4, 14–16, 21, 29, 30] have been created for automatically partitioning applications using program analysis. These tools' partitioning algorithms, however, consider only security but not other factors; as a result, they do not allow tradeoffs among multiple factors. Further, these tools partition programs, but lack consideration for helping users achieve security goals; e.g., they expect users to manually choose where to declassify information flows and sometimes do not account for flows to external channels.

ProgramCutter [28], Swift [6], and SOAAP [12] 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 [28] 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. For example, if function A reads a password from a password file and calls function B with that password, ProgramCutter would not label function B as sensitive since B does not directly read the file. In contrast, PM tracks how function A propagates the password inside A and determines how many bits of sensitive information are passed from A to B.

Swift [6] 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 [19] 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).

[1] Gerrymandering refers to the process of manipulating the boundaries of voting districts to favor one political party; program-mandering refers to the process of carefully choosing partitioning boundaries to favor a good tradeoff between security and performance.

SOAAP [12] 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. In comparison, PM asks for only annotations about sensitive data together with security/performance budgets, and it automatically finds a partition boundary that satisfies the budgets and is optimized for one metric. 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.

Dong et al. [9] performed an empirical study on balancing between security and performance for a privilege-separated web browser. Using past vulnerability and profiling data, the case study quantifies the security and performance benefits for typical kinds of privilege separation in the context of web browsers. However, it does not offer a framework that allows users to make security and performance tradeoffs to privilege separate general applications.

PM partitions programs to control the information flow between the sensitive and the insensitive domains. DataShield [5] separates sensitive and non-sensitive data and applies memory safety on sensitive data and Software-based Fault Isolation [26, 27] on non-sensitive data; no information flow is allowed between the two memory regions. Although DataShield instruments each memory instruction according to its privilege (whether sensitive data is accessed) and enforces a logical separation of code, it does not split code into two separate domains. The danger of mixing code of different privileges in the same domain is that the domain still has to possess all privileges; the attacker might use a vulnerability to execute code of higher privileges unexpectedly. In contrast, privilege separation tools physically separate code into multiple domains of differing privileges. Kenali [25] is similar to DataShield, except that it works on OS kernel code, and applies data-flow integrity on kernel data critical for access control.

```
1  char* fname[5]={"/d1/pwd", "/d2/pwd", ...};
2
3  // return -1 if fn does not exist;
4  // return 1 if userpwd found in fn;
5  // return 0 if userpwd not found in fn
6  int auth2 (char* userpwd, char* fn){
7    ... // code omitted
8  }
9
10 int auth (char* userpwd){
11   for (int i = 0; i < 5; i++){
12     int ret = auth2(userpwd, fname[i]);
13     if (ret != -1) return ret;
14   }
15   return -1;
16 }
17
18 void main (int argc, char **argv){
19   char *username=argv[1], *pwd=argv[2];
20   char userpwd[500];
21   //possible buffer overrun
22   sprintf(userpwd, "%s %s", username, pwd);
23   int ret = auth(userpwd);
24   if (ret==1) printf("Auth succeeded!\n");
25   else if (ret==0) printf("Auth failed!\n");
26   else printf("Password file not found!\n");
27 }
```

Figure 1: A motivating example.

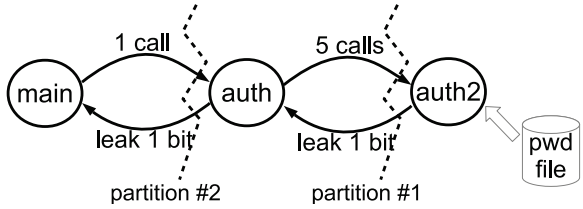


Figure 2: Call graph and partitions for Fig. 1 example.

3 A MOTIVATING EXAMPLE

Fig. 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 `thttpd` 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, 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 Fig. 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`.

Fig. 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 systems that require declassification [4, 15], 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 previous systems [4, 15], 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. Some earlier work [4, 15] 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 SYSTEM OVERVIEW

Attack model. PM takes a program with some sensitive data and splits it into a sensitive and an insensitive domain. Sensitive data can be data of either high secrecy or low integrity. We assume programmers can identify such sensitive data and are able to annotate the program to make sensitive data explicit. As we will explain, for both secrecy and integrity, PM's overall security goal of partitioning is based on information flow, particularly noninterference [11]. However, what the attacker controls in the case of secrecy differs from the one for integrity. We next explain them separately (and leave more details to Sec. 6).

When the goal is to prevent leakage of secret data, users use PM to produce a sensitive, high-secrecy domain that holds and processes a secret and an insensitive, low-secrecy domain for the rest of the program. Here, we assume a strong attack model in which the attacker can fully control the low-secrecy domain by modifying its data and possibly its code or control flows (e.g., via code injection or return-oriented programming); she can also control the interaction from the low-secrecy domain to the high-secrecy domain given the partition boundary. Given the attacker's capabilities, PM aims to produce a partition that achieves non-interference: the secret data does not leak to the low-secrecy domain, modulo the data that is declassified by the high-secrecy domain at the boundary. PM does not determine how to declassify sensitive data automatically, but enables assessment of the leakage rate for information flows at boundaries to help users choose boundaries for implementing declassifiers.

When the goal is to confine the use of low-integrity data, users apply PM to produce a sensitive, low-integrity domain that processes low-integrity data (e.g., untrusted data from the internet) and an insensitive, high-integrity domain for the rest of the program. In this case, we assume the low-integrity (sensitive) domain can be fully controlled by the attacker because of the low-integrity data (in contrast to the assumption that the *insensitive* domain can be fully controlled in the case of secrecy). The attacker's goal is to influence the execution of the high-integrity domain through controlling the interactions between the low-integrity to the high-integrity domains. PM aims to produce a good partition that also achieves non-interference: the low-integrity data cannot affect the execution of the high-integrity domain, modulo the data that is endorsed at the boundary from the low-integrity to the high-integrity domain. Similarly to secrecy as described above, PM aids users in the selection of partition boundaries for implementing endorsers.

System workflow. Fig. 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. We reuse our previous work [15] for PDG construction; detailed algorithms can be found there. The user also annotates the application to tell PM what the sensitive data is. PM then performs program analysis to quantify security and performance using selected metrics. For instance, it uses a dynamic information-flow tracker to measure the quantity of sensitive flow among functions and global variables. Those measurements are used to annotate the nodes and edges of the PDG. Based on the annotated PDG and user-specified constraints on the values of metrics, a partitioning algorithm searches for a partition that satisfies the constraints and is optimal according a user-specified goal (i.e., one of the metrics). The output is a sensitive domain and an insensitive domain, each of which consists of both data and code. In principle, this approach can be applied multiple times to further decompose the resultant partitions.

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

5 GRAPH-BASED PARTITIONING

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

5.1 Graph partitioning

For ease of exposition, we present the formalization in two steps. First, we assume the input program consists of a set of functions and has no global variables; here we formalize program partitioning as call-graph partitioning. In the second step, we consider the case when the program has both functions and global variables; in this step, program partitioning is modeled as partitioning a program-dependence graph.

We model a program that has a set of functions but no global variables as a call graph. It is a directed graph $G = (V, E)$, with vertices V representing the program's functions and edges E representing call relations between functions. If function f_1 can call

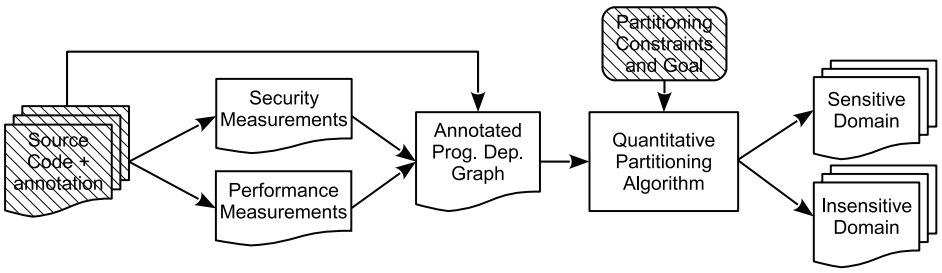


Figure 3: System flow of PM.

f_2, there is a call edge from f_1 to f_2.[2] Without loss of generality, we assume there is a single special function that reads in sensitive data; we use s for the special function. For the example in Fig. 1, `auth2` is the special function since it reads the password file.

Definition 5.1 (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, `thttpd` has a function called `my_snprintf`, a custom implementation of `snprintf`. It is called by many functions; without duplication, it would cause many domain crossings no matter what domain `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 $FB = \{e_{ij} \mid i \in S \wedge j \in T - R\}$. Backward boundary edges are those from T to $S - R$, defined as $BB = \{e_{ij} \mid i \in T \wedge j \in S - R\}$. 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 6. We discuss in Sec. 6.2 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 5.2 (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. Given the above understanding, Appendix A 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 our previous work [15] to construct full-fledged PDGs [10], 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.

5.2 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.

[2] This formalization has just forward edges but no return edges; measurements for both the call and the return are associated with forward edges. This makes the formalization easier to present and implement.

Appendix B presents in detail how to encode optimal partition as a binary IP problem for the metrics we introduce in Sec 6. 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.

6 METRICS

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. [20] 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 [23], approximating Clark-Wilson integrity [8]. 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.

In this section, we examine the information-flow security requirements that must be fulfilled in order to deploy sensitive domains that process low-integrity data and high-secrecy data. Measuring these requirements form a basis for key security metrics used by PM to generate partitions. We then define the full set of security metrics applied in our evaluation using PM.

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.

6.1 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, 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 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 fflow(e) and 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 6.1 (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 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 6.2 (Sensitive code percentage).

$$\text{SCode}(S,T) = (\sum_{v \in S} \text{sz}(v))/(\sum_{v \in S \cup T} \text{sz}(v)).$$

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 6.3 (Context switch frequency).

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

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 [15] 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 [2]); 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 paper, 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 `int` is 0, the complexity of `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 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 6.4 (Pointer complexity).

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

6.2 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 paper, 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.

7 IMPLEMENTATION

The prototype implementation of PM is implemented with the help of several tools, including LLVM[3], Flowcheck [18], Intel's Pin tool [17], and `lp_solve` [1]. We first explain the toolchain's input and output, followed by a discussion of how each component is implemented.

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.

[3]Our original implementation was on LLVM 3.5; we recently migrated the PDG-construction part to LLVM 5.0 and open sourced it (https://bitbucket.org/psu_soslab/pdg-llvm5.0/src/pdg_plugin/). Other parts of the tool are being migrated to LLVM 5.0 and will be released when mature.

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

```
int (__attribute__((annotate("sensitive"))) auth2)
 (char* userpwd, char* fn) { ... }
```

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.

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 our previous work [15], 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 [18] 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 C 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 frequency when a call edge or a data-flow edge in the PDG becomes a boundary edge after partitioning, PM uses Intel's Pin tool [17] 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 B. 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.[4] In detail, the

[4] 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

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 `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.

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). `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.

8 EXPERIENCE REPORT

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, (_*, _, _, _) 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:

be able to plug it into our tool and remove the single-directional RPC constraints from the IP model.

Program	SLOC	Func.	Glob.	Sensitive data; Annot.
chsh (3.3.2)	564	8	10	pwd file; 1
chage (3.3.2)	948	14	25	pwd file; 1
passwd (3.3.2)	1,168	14	33	pwd&shadow files; 1
useradd (3.3.2)	2,395	24	49	pwd&shadow files; 4
telnet (1.9.4)	11,120	180	139	data from internet; 3
thttpd (2.25)	11,403	145	128	authentication file; 1
wget (1.18)	61,217	666	176	data from internet; 29
nginx (1.9.5)	160,293	1,064	422	authentication file; 1

Table 1: Benchmarks used during evaluation.

- *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 (_*, 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, increases 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 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 a recent system PtrSplit [15],

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, and 16GB of physical memory.

8.1 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. In fact, our toy program used in Sec. 4 is inspired by this authentication pattern.

Partitioning process. In *thttpd*, `ac2` is the only function that interacts with the password file. A natural partition would be to put only `ac2` into the sensitive domain. However, it would require bidirectional RPC support as `ac2` invokes several other functions. Since only single-directional RPCs are supported, this choice is not feasible. Therefore, we used PM to explore partitioning choices of *thttpd* and Table 2 includes a summary of the result. For each partition, the table lists the budgets we used, the amount of IP-solving time to produce the optimal partition, the quality scores of the produced partition, and also the runtime overhead of the partitioned *thttpd* when downloading files. Next we discuss in detail about how these partitions were produced.

We started with unlimited budgets and optimized for the smallest sensitive domain: $(\_^*, \_, \_, \_)$, for which PM produced partition ① in Table 2. This partition has the smallest sensitive domain among all single-directional choices; further, it leaks only one bit of information, common during authentication. This initial partition could be acceptable, but we used PM further to make tradeoffs, in an attempt to reduce performance overhead.

By following tradeoff strategy 1, we chose the context-switch frequency as the target and the sensitive-code percentage as the sacrifice and revised the previous partition's quality (9.15%, 1.0, 1455.6, 9.0) to the new budgets $(\_^*, 1.0, 1455.5, 9.0)$. PM found a partition with quality (9.27%, 1.0, 1411.2, 8.0), shown as partition ② in Table 2. Further improvements can be made by repeating the same strategy. For example, with one more step based on partition ②'s quality metrics, budgets $(\_^*, 1.0, 1411.1, 8.0)$ produced another tradeoff choice shown as partition ③ in the table.

Alternatively, by following tradeoff strategy 2 on the quality metrics of partition ①, we gave budgets $(10.00\%, 1.0, 1455.5^*, 9.0)$. PM happened to produce the same partition as partition ③.

We inspected *thttpd*'s source code to understand the three partitioning choices produced by PM. Partition ① separates `ac` and `ac2` to maintain the smallest sensitive domain. Partition ② further adds a logging function into the sensitive domain to reduce the number of context switches. Partition ③ cuts at a higher execution level, which separates `ac` and its caller, since `ac` is executed less frequently than `ac2` in typical situations. In all, the three partitions are reasonable choices that would be produced by manual partitioning when making tradeoffs between security and performance.

Assessing security and performance. In terms of security, all three choices separate `ac2` into the sensitive domain. After inspection, we determined that the sensitive-domain code in all three choices only reads from the password file. Furthermore, the code does not write to the file system using any I/O operation; thus it cannot leak secret passwords through the file system. As a result, the only sensitive information that can be leaked is the one bit of authentication response, acceptable during authentication.

For evaluating performance, we implemented the three partitions and experimented with different settings. The most realistic setting is to download moderate-sized (1M) files from a remote *thttpd* server as this matches its typical use case. However, for completeness, we also evaluated other settings, including downloading small-sized (1K) files and downloading from a local *thttpd* server (on the same machine as the client). Table 2 presents the runtime overhead of partitioned *thttpd* in different settings, when compared with unpartitioned *thttpd*. In general, all three choices have small overheads for the typical case of downloading moderate-sized files from a remote server. Partition ③ has significantly less overhead compared to the other two, justifying the benefit of performing iterative refinement via PM. Further, the results show that our metrics of context-switch frequency and pointer complexity positively correlate with performance overhead.

Comparison with PtrSplit. PtrSplit also partitioned *thttpd* and produced one single choice with the help of manual declassification. Its partitioning result is similar to the third choice in Table 2, except that it simplified *thttpd*'s functionality to remove logging and accessing to remote global variables. In contrast, our partitions support both. To make a fair comparison, we augmented PtrSplit's result with the ability of logging and remote global variable accessing and experimented with this new partition. The overhead data for the remote server case is 0.5% (1M files) and 35.7% (1K files); for the local server case, it is 28.4% (1M) and 36.5% (1K). It has a higher overhead than our partition ③ because our partition puts a logging function into the sensitive domain. PtrSplit's partition puts the logging function in the insensitive domain; so it has to invoke an RPC function to access some global data in the sensitive domain.

8.2 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

	Budgets (b_c, b_f, b_s, b_x)	IP-Solving Time (s)	SCode(%)	Flow	CSwitch	Cplx	Overhead(%) (FileSize: 1M/1K)	
							Remote	Local
①	(_*, _, _, _)	0.10	9.15	1.0	1455.6	9.0	1.2/54.7	37.1/63.1
②	(_*, 1.0, 1455.5, 9.0)	0.21	9.27	1.0	1411.2	8.0	1.1/51.1	35.5/60.9
③	(_*, 1.0, 1411.1, 8.0)	0.18	9.62	1.0	1400.1	8.0	0.4/34.2	25.1/30.9

Table 2: Partitioning choices for *thttpd*.

	Budgets (b_c, b_f, b_s, b_x)	IP-Solving Time (s)	SCode(%)	Flow	CSwitch	Cplx	Overhead(%) (FileSize: 1M/1K)	
							Remote	Local
①	(_*,_,_,_)	0.80	11.03	4047.0	1213.8	117.0	1493.0/6.2	13799.0/13.4
②	(50.00%, 999.0*, 38.2, _)	2.03	49.12	8.0	38.2	45.0	1.6/1.9	6.4/2.1
③	(16.00%, _, _*, _)	1.13	15.68	4052.0	198.5	137.0	412.0/7.2	1440.0/7.9
④	(_*, 2.0, 38.2, _)	1.56	78.42	2.0	38.2	14.0	1.5/2.3	7.6/3.3

Table 3: Partitioning choices for *wget*.

data contained potentially malicious data (i.e., low-integrity data). The goal of partitioning is to produce a sensitive, low-integrity domain that interacts with the server, and an insensitive domain for the rest of the program. The goal is to protect the insensitive domain from compromise due to untrusted file input retrieved from servers, enabling sandboxing of the file download.

Partitioning process. During partitioning, we marked all functions that interact with the internet as sensitive. We started with the smallest sensitive domain by using budgets (_∗, _, _, _) and after a simple iteration we found an initial partition with quality (11.03%, 4047, 1213.8, 117.0), which is shown as partition ① in Table 3. 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 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 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 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.

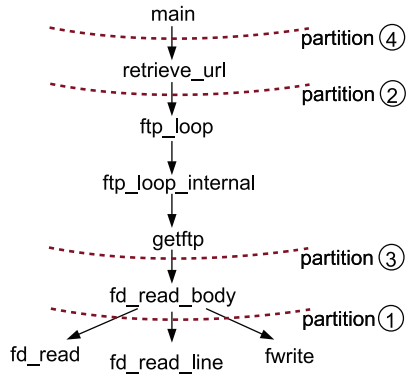


Figure 4: Call graph and partitions for *wget*.

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 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.

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.

8.3 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. During partitioning, we first marked functions `process_rings`, `netflush`, and `tn` as sensitive, 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. 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.

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. 5.

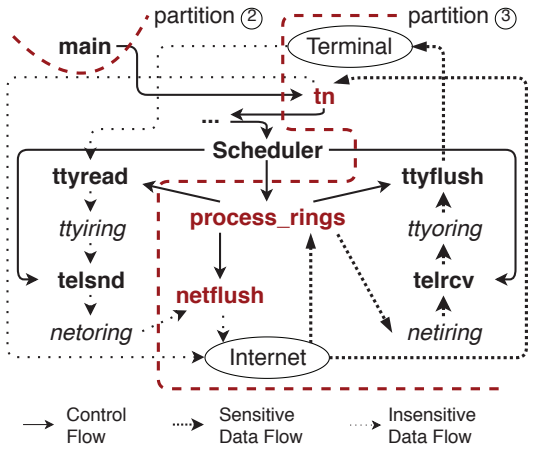


Figure 5: PDG and partitions for *telnet*.

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.

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.

8.4 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. 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 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

	Budgets (b_c, b_f, b_s, b_x)	IP-Solving Time (s)	SCode(%)	Flow	CSwitch	Cplx	Overhead(%)	
							Remote	Local
①	(_*,999,_,_)	0.41	74.11	3.0	609.0	146.0	N/A	N/A
②	(_*,999,16.0,15.0)	2.30	86.32	0.0	16.0	15.0	7.9	34.33
③	(_*,999,_,_)	0.21	13.10	26.0	13305.0	227.0	N/A	N/A

Table 4: Partitioning choices for *telnet*; "N/A" are for partitions that were not implemented.

	Budgets (b_c, b_f, b_s, b_x)	IP-Solving Time (s)	SCode(%)	Flow	CSwitch	Cplx	Overhead(%)	
							Remote	Local
①	(_*,999,_,_)	2.67	2.89	1.0	136.2	437.0	N/A	N/A
②	(_*,999,_,_)	N/A	2.89	1.0	14.0	32.0	34.1	21.4

Table 5: Partitioning choices for *nginx*; "N/A" are for partitions that were not implemented.

Program	Version	Vulnerability	Mitigated by
thttpd	2.25	CVE-2013-0348	① ② ③
		CVE-2009-4491	① ② ③
		CVE-2006-4248	① ② ③
wget	1.18	CVE-2018-0494	② ④
		CVE-2017-6508	④
		CVE-2017-13090	② ③ ④
		CVE-2017-13089	② ④
telnet	1.9.4	CVE-2005-0468	① ② ③
		CVE-2005-0469	① ② ③
		Exploit-DB-45982[5]	① ② ③
nginx	1.9.5	CVEs from 2016 to 2018[6]	②

Table 6: Mitigated vulnerabilities by different partitions.

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.

8.5 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 7. Note that these programs are small, which excluded us from finding multiple interesting partitions.

Prog	SCode (%)	Flow	CSwitch	Cplx	Overhead (%)
chsh	51.52	0.0	0.5	2.0	1.00
useradd	50.94	0.0	1781.4	29.0	11.33
passwd	82.33	0.0	846.2	13.0	7.50
chage	6.57	0.0	77.0	2.0	80.63

Table 7: Partitioning choices for *chsh* and *useradd*

8.6 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 security metric is the amount of past known vulnerabilities (e.g., used in [9]) 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 6 lists all CVEs for the versions of programs we used, and whether a CVE can be mitigated by a partition produced by PM.

[5] multiple overflows https://www.exploit-db.com/exploits/45982

[6] including CVE-2018-16845, CVE-2018-16844, CVE-2018-16843, CVE-2017-7529, CVE-2016-0747, CVE-2016-0746,CVE-2016-0742, and CVE-2016-4450.

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. 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 6 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

9 DISCUSSION AND FUTURE WORK

We discuss limitations of PM, some of which were discovered during evaluation, and how it can be extended to address them. First, similar to other tools, PM performs 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., `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.

Second, currently PM partitions a program into two domains according to a security lattice of two points. We plan to extend it to support more complex lattices that can result in more than two domains (e.g., mixing confidentiality and integrity), such as what Swift [6] does. This introduces the complication of allowing further code duplication at different security levels.

Third, PM's optimization framework currently supports only one optimization metric. A natural alternative to having four metrics would be to weight each of them 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 is a framework that automatically produces information about how to partition a program, but does not offer complete automation in implementing a partition. Given information about a partition, a user needs to manually split files and adjust compilation scripts. This can be labor intensive, which was the major reason why we tested only a few partitions for programs during evaluation. Automating these steps is feasible, but requires additional engineering effort.

Fifth, PM's implementation relies 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 [24] for a recent survey) have been theoretical and not produced practical tools. For instance, Clark et al. [7] described a static analysis that over-approximates 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.

Finally, by generating the optimal partition, PM automatically computes 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.

10 CONCLUSIONS

We have proposed PM, 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 experience with real applications suggests 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 that balances security and performance better.

11 ACKNOWLEDGMENTS

We thank anonymous reviewers and our shepherd, Lorenzo DeCarli, for their insightful comments. This research is based upon work supported by US NSF grants CNS-1801534, CCF-1723571, CNS-1408826, CNS-1816282, CNS-1408880, CNS-1526319, 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.

REFERENCES

[1] 2016. lp_solve 5.5 reference guide. http://lpsolve.sourceforge.net/.
[2] David M. Beazley. 1997. *SWIG Users Manual: Version 1.1.*
[3] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation.* 309–322.

[4] David Brumley and Dawn Song. 2004. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *13th Usenix Security Symposium*. 57–72.
[5] Scott A. Carr and Mathias Payer. 2017. DataShield: Configurable Data Confidentiality and Integrity. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. 193–204.
[6] Stephen Chong, Jed Liu, Andrew Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. 2007. Secure Web Applications via Automatic Partitioning. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. 31–44.
[7] David Clark, Sebastian Hunt, and Pasquale Malacaria. 2007. A static analysis for quantifying information flow in a simple imperative language. *Journal of Computer Security* 15, 3 (2007), 321–371.
[8] David D. Clark and D. R. Wilson. 1987. A Comparison of Commercial and Military Computer Security Policies. In *IEEE Symposium on Security and Privacy (S&P)*. 184–195.
[9] Xinshu Dong, Hong Hu, Prateek Saxena, and Zhenkai Liang. 2013. A Quantitative Evaluation of Privilege Separation in Web Browser Designs. In *18th European Symposium on Research in Computer Security (ESORICS)*. 75–93.
[10] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems* 9, 3 (July 1987), 319–349.
[11] Joseph A. Goguen and José Meseguer. 1982. Security Policies and Security Models. In *IEEE Symposium on Security and Privacy (S&P)*. 11–20.
[12] Khilan Gudka, Robert N. M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. 2015. Clean Application Compartmentalization with SOAAP. In *22nd ACM Conference on Computer and Communications Security (CCS)*. 1016–1031.
[13] Douglas Kilpatrick. 2003. Privman: A library for partitioning applications. In *USENIX Annual Technical Conference, FREENIX track*. 273–284.
[14] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter R. Pietzuch. 2017. Glamdring: Automatic Application Partitioning for Intel SGX. In *USENIX Annual Technical Conference (ATC)*. 285–298.
[15] Shen Liu, Gang Tan, and Trent Jaeger. 2017. PtrSplit: Supporting General Pointers in Automatic Program Partitioning. In *24th ACM Conference on Computer and Communications Security (CCS)*. 2359–2371.
[16] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In *22nd ACM Conference on Computer and Communications Security (CCS)*. 1607–1619.
[17] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM Conference on Programming Language Design and Implementation (PLDI)*. 190–200.
[18] Stephen McCamant and Michael D. Ernst. 2008. Quantitative information flow as network flow capacity. In *ACM Conference on Programming Language Design and Implementation (PLDI)*. 193–205.
[19] Andrew Myers and Barbara Liskov. 2000. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering Methodology* 9 (Oct. 2000), 410–442. Issue 4.
[20] Niels Provos, Markus Friedl, and Peter Honeyman. 2003. Preventing privilege escalation. In *12th Usenix Security Symposium*. 231–242.
[21] Konstantin Rubinov, Lucia Rosculete, Tulika Mitra, and Abhik Roychoudhury. 2016. Automated partitioning of Android applications for trusted execution environments. In *International Conference on Software engineering (ICSE)*. 923–934.
[22] Jerome Saltzer and Michael Schroeder. 1975. The Protection of Information in Computer Systems. *Proceedings of The IEEE* 63, 9 (Sept. 1975), 1278–1308.
[23] Umesh Shankar, Trent Jaeger, and Reiner Sailer. 2006. Toward Automated Information-Flow Integrity Verification for Security-Critical Applications. In *Network and Distributed System Security Symposium (NDSS)*. 267–280.
[24] Geoffrey Smith. 2015. Recent Developments in Quantitative Information Flow (Invited Tutorial). In *IEEE Symposium on Logic in Computer Science (LICS)*. 23–31.
[25] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. 2016. Enforcing Kernel Security Invariants with Data Flow Integrity. In *Network and Distributed System Security Symposium (NDSS)*.
[26] Gang Tan. 2017. Principles and Implementation Techniques of Software-Based Fault Isolation. *Foundations and Trends in Privacy and Security* 1, 3 (2017), 137–198.
[27] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. 1993. Efficient Software-Based Fault Isolation. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. ACM Press, New York, 203–216.
[28] Yang Liu Yongzheng Wu, Jun Sun and Jin Song Dong. 2013. Automatically partition software into least privilege components using dynamic data dependency analysis. In *International Conference on Automated Software Engineering (ASE)*. 323–333.
[29] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew Myers. 2002. Secure program partitioning. *ACM Transactions on Compututer Systems (TOCS)* 20, 3 (2002), 283–328.
[30] Lantian Zheng, Stephen Chong, Andrew Myers, and Steve Zdancewic. 2003. Using Replication and Partitioning to Build Secure Distributed Systems. In *IEEE Symposium on Security and Privacy (S&P)*. 236–250.

A 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, $RE = \{e_{ij} \mid i \in GV \ \wedge \ j \in FV\}$ for the set of read edges, and $WE = \{e_{ij} \mid i \in FV \ \wedge \ j \in GV\}$ 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 $FB_C = \{e_{ij} \in CE \mid i \in S \ \wedge \ j \in T - R\}$; (2) forward boundary read edges $FB_R = \{e_{ij} \in RE \mid i \in S - R \wedge j \in T\}$; (3) forward boundary write-edges $FB_W = \{e_{ij} \in WE \mid i \in S \ \wedge \ j \in T - R\}$. 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 $BB_C = \{e_{ij} \in CE \mid i \in T \wedge j \in S - R\}$; (2) backward boundary read edges $BB_R = \{e_{ij} \in RE \mid i \in T - R \ \wedge \ j \in S\}$; (3) backward boundary write edges $BB_W = \{e_{ij} \in WE \mid i \in T \ \wedge \ j \in S - R\}$. We have $BB = BB_C \cup BB_R \cup BB_W$.

Furthermore, weights presented in Sec. 6 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: $\text{fflow}(e)$ is the amount of information flow on the edge and $\text{bflow}(e)$ is always zero.

Third, for a read edge e from global g to function f, we use $\text{af}(e)$ for the frequency of f reading from g during profiling; similarly, for a write edge e from function f to global g, $\text{af}(e)$ 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, $\text{plevel}(e)$ 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.

B ENCODING OPTIMAL PARTITIONING AS IP

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

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

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 $T - R$:

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

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, $x_{ij} = 1 \Leftrightarrow e_{ij} \in FB$; and $y_{ij} = 1 \Leftrightarrow e_{ij} \in BB$.

With the input budgets b_c, b_f, b_s, and b_x, we can construct the following constraints for all measurements based on Def 5.2:

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

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

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

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

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 α_i, α_j, β_i, and β_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,

$$x_{ij} = 1 \ \Leftrightarrow \ e_{ij} \in FB_C \ \Leftrightarrow \ \beta_i = 0 \wedge \beta_j = 1 \ \Leftrightarrow \ \neg\beta_i \wedge \beta_j;$$
$$y_{ij} = 1 \ \Leftrightarrow \ e_{ij} \in BB_C \ \Leftrightarrow \ \alpha_i = 0 \wedge \alpha_j = 1 \ \Leftrightarrow \ \neg\alpha_i \wedge \alpha_j;$$

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

$$x_{ij} = 1 \ \Leftrightarrow \ e_{ij} \in FB_R \ \Leftrightarrow \ \alpha_i = 1 \wedge \alpha_j = 0 \ \Leftrightarrow \ \alpha_i \wedge \neg\alpha_j;$$
$$y_{ij} = 1 \ \Leftrightarrow \ e_{ij} \in BB_R \ \Leftrightarrow \ \beta_i = 1 \wedge \beta_j = 0 \ \Leftrightarrow \ \beta_i \wedge \neg\beta_j;$$

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

$$x_{ij} = 1 \ \Leftrightarrow \ e_{ij} \in FB_W \ \Leftrightarrow \ \beta_i = 0 \wedge \beta_j = 1 \ \Leftrightarrow \ \neg\beta_i \wedge \beta_j;$$
$$y_{ij} = 1 \ \Leftrightarrow \ e_{ij} \in BB_W \ \Leftrightarrow \ \alpha_i = 0 \wedge \alpha_j = 1 \ \Leftrightarrow \ \neg\alpha_i \wedge \alpha_j.$$

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 $y = 1 \ \Leftrightarrow \ x_1 \wedge x_2 \wedge \cdots \wedge x_n$ can be linearly modeled as

$$\begin{aligned} y &\leq x_i, \ \forall i = 1, 2, \ldots, n \\ y &\geq x_1 + x_2 + \cdots + x_n - (n - 1). \end{aligned}$$

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

$$\begin{aligned} x_{ij} &\leq 1 - \beta_i, \\ x_{ij} &\leq \beta_j, \\ x_{ij} &\geq \beta_j - \beta_i, \\ y_{ij} &\leq 1 - \alpha_i, \\ y_{ij} &\leq \alpha_j, \\ y_{ij} &\geq \alpha_j - \alpha_i. \end{aligned}$$

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

C 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 := ebx xor ecx
...
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 := ebx xor ecx
...
eax := edx
...
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;[7] (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.

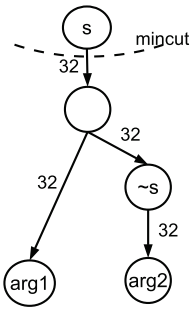
We note that the rewriting is performed purely for measuring sensitive information flow. After the measurement, the rewritten program is discarded and PM's partitioning is performed on the original program.

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. 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.

[7] The "not" instruction in x86 is like C's one's complement (~) operation, but not C's logical not (!) operation.



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 fflow(e) is the sum of forward flow quantities in multiple dynamic calls that correspond to the same call edge e; the same goes for bflow(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.