Automated Program Hardening via Hoisted Privilege Reductions

by

Shreeasish Kumar

B.Tech., KIIT University, 2017

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
Department of Applied Sciences
Faculty of Computing Science

SIMON FRASER UNIVERSITY
Fall 2019

Approval

Name:	**Shreeasish Kumar**
Degree:	**Master of Science (Computer Science)**
Title:	**Automated Program Hardening via Hoisted Privilege Reductions**
Examining Committee:	**Chair:** Arrvindh Shriraman Professor
	William (Nick) Sumner Senior Supervisor Associate Professor
	Keval Vora Supervisor Assistant Professor
	Anoop Sarkar Examiner Professor School of Computing Science
Date Defended:	**September 24, 2019**

Abstract

Privilege based security policies for programs are effective as a first line of defense against attacks. They are able to mitigate broad classes of attacks against programs, potentially saving the costs of searching for and mitigating specific vulnerabilities. Deploying such techniques, however requires expert knowledge and manual analysis of programs.

We propose Passive Privilege Inference and Reducer (PPIR), a technique driven by a novel static analysis that automates the process of inferring the privileges required by a program. We develop a tool that uses this technique to infer the privileges required by a program and instrument it with a security policy to enforce the Principle of Least Privilege. We show that PPIR performs on par with handcrafted security measures while eliminating the manual burden of investigating and inserting privileges. PPIR further enables the potential to progressively reduce privileges as a program executes.

Keywords: Static Analysis; Code Hoisting; Privilege Restriction

Dedication

Dedicated to Baba, Ma and Babai

Acknowledgements

I will be forever grateful for the unerring guidance and unwavering support from Dr. William Sumner; without his patience, this thesis would not be possible.
Critical to this thesis, were the kind words and encouragement from my brother, Dr. Snehasish Kumar. I wouldn't be able to keep moving forward without them.
Finally, I would like to thank my friends and family for their help and support.

Table of Contents

List of Tables

List of Figures

Listings

List of Algorithms

Chapter 1

Introduction

Software security is an increasingly important part of software development[26] where software that is already built must be protected. Permission based security models are widely used today[6], particularly in mobile operating systems such as Android and iOS. Within this security model, only the applications holding permissions for a resource are authorized to access that resource, e.g. the camera. The permissions needed by a program are selected by the developer and declared to the operating system. Once authorized, the protected resource can be used at any point during the program's execution. Such static permissions are easy to use but can produce programs that are *over-privileged*, i.e. have more permissions than necessary[1][14][5].

Dynamic security models permit modifications to the permissions or *privileges* available to a program. Privileges can be revoked over the course of a program's execution or selectively granted to distinct parts of the program. Such models offer finer control over the security of a program. The Principle of Least Privilege[33] dictates a program should execute with the smallest set of privileges necessary to complete its task. Using the Principle of Least Privilege as a guide, developers can exercise the finer control offered by dynamic security models to more effectively secure applications in comparison to static security models[34].

The Pledge framework[29], as part of the OpenBSD, distribution allows dynamic privilege restrictions, i.e. developers can choose to remove the privileges available to an application at pre-specified points during execution. A privilege once removed, cannot be recovered by the program. Thus, privilege restrictions are *monotonic*. A program attempting to invoke a system call that it does not have privileges for is terminated by the operating system. In practice enforcing privilege restrictions via Pledge can be a significant task for the developer due to the following factors:

1. The privilege requirements of the program may not easily map to the available privileges specifications. Libraries used by developers must be manually analyzed to extract their privilege requirements.

2. Pledge allow developers to monotonically restrict privileges but identifying which privileges to remove is complicated by control flow. Programs may invoke different system calls on different paths in the program, which leads to differing privilege requirements across different branches.

3. The privilege requirements of the program may change as the source code evolves. Programs may add new functionality that requires new privileges, necessitating a reanalysis of the privilege requirements of the program.

To ease the use of privilege restrictions, we develop Passive Privilege Inference and Reducer (PPIR), an automated inference system which leverages the Pledge framework to perform privilege management guided by the Principle of Least Privilege. 'Passive' indicates that the tool does not require any interaction or input from the user besides the source code. PPIR instruments source code to dynamically restrict privileges at run-time based on a novel static analysis technique. We demonstrate the efficacy of our tool on real world programs typical to any minimal Unix based distribution. While our tool is built using the Pledge framework, the underlying insights and static analysis can be translated to other privilege management systems that support monotone privilege restrictions.

Chapter 2

Background and Motivation

The Principle of Least Privilege[33] acts as a guide for minimizing the possible damage arising from an exploit, the idea being that the software should only have the smallest set of privileges needed to complete its task. Software can use security features offered by the operating system[21][39][37] to enforce the Principle of Least Privilege.

The Pledge framework offered with the OpenBSD operating system defines a privilege as a group of system calls that have roughly the same effect[29]. The Principle of Least Privilege guides developers to remove privileges when unnecessary. When using the Pledge framework, developers should determine points in the program after which system calls sharing a particular privilege will not be called. At that point, that privilege is unnecessary for the program to continue execution and can be removed from the set of privileges held by the program. Using the Pledge framework offers developers the ability to voluntarily remove privileges with a call to pledge(char*). After a privilege is removed, if the program is compromised by an attacker, the attacker will be limited to the privileges available to the program.

The pledge function accepts a string as its argument. The first argument is the list of privileges that are to be held by the program. This is enforced by the operating system. A process starts with a set of enabled permission bits corresponding to the privileges defined by the Pledge framework. A call to pledge removes the bits corresponding to the privileges which are *not* specified in the first string argument to pledge. Once a permission bit is disabled i.e. a privilege is removed, it cannot be enabled again. Thus, processes can only give up privileges once they are unnecessary. When the process invokes a privilege using system call, the operating system checks whether the appropriate bits are still enabled for the system call. Processes failing this check are terminated and an error log is generated for the user.

Summarily, a process starts with access to the full set of privileges and can invoke calls to pledge to give up privileges during its execution. The Pledge framework also supports restricting privileges for any sub-processes that may be spawned. This work focuses on programs that execute within a single process and this is feature is not considered. For our

threat model, we consider attackers that can feed inputs to a program, potentially leading to unintended behaviour. The Pledge framework aims to mitigate the damage done by such an attacker by removing unnecessary privileges which can limit the damage done by an attack.

```
1  pledge("stdio rpath wpath")
2  FILE* fp = fopen("input.txt", "r");
3  // Requires rpath
4  ...
5  pledge("stdio wpath");
6  ...
7  if (ENABLE_WRITE) {
8      FILE* fp2 = fopen("output.txt", "w");
9       // Requires wpath
10     fputs(data, fp2);
11 }
12 pledge("stdio");
13 printf("Printing Output: %s", data);
14  // Requires stdio
15 ...
```

Listing 2.1: Progressive Privilege Reduction

Monotonic Privilege Restriction Each privilege defined by the Pledge framework is shared by several system calls. The "stdio" privilege is required by each system call that intends to read or write to open file streams or to standard IO. The "rpath" privilege is required by programs to open files for reading exclusively. Similarly, the "wpath" privilege is required to open files for writing. Note that, a file stream once open can be written to or read from without the "rpath" or "wpath" privilege. Ideally, a process would progressively give up privileges after the last system call that requires the privilege, i.e. as soon as the privilege is no longer required. Figure 2.1, shows an example where a program reads from one file and writes to another. Line 1 is a call to pledge() that restricts the privileges to "stdio rpath wpath". After this point, the program will not be able use any system calls aside from those associated with the stdio, rpath and wpath privileges. The string passed to pledge tells the operating system which privileges are to be retained. After the program has opened the file in read mode (line 2), it no longer requires the rpath privilege and removes it with a call to pledge on line 5. The string passed to pledge only holds wpath and stdio which removes the rpath privilege and prevents the program from opening any other files in read mode. Similarly, once the program has finished writing to the file on line 10, it removes the privilege for writing to files. stdio is retained since it is required later in the program outside of the example. We use the term *monotonic privilege reduction* to define the paradigm in

which developers may progressively reduce the privileges available to the program via calls to pledge.

Compared to other frameworks such as Capsicum[39] or SELinux[37] that require the program to split into isolated units, the Pledge framework does not require the program to be refactored. The Pledge framework has also been demonstrated on 50 applications prior to its adoption by the OpenBSD community. The burden of performing privilege restrictions, however, still lies with the developer. Performing monotonic privilege restrictions requires the developer to identify the exact privilege requirements of the program as well as when a privilege is no longer required. This requires reasoning over every path and function in the program.

System calls that require privileges may be located on multiple paths over multiple functions. The developer must ensure that all privilege requirements induced by system calls are accounted for prior to inserting a call to pledge. Similarly the developer must also locate the last use of a privilege, which may be shared by several system calls on multiple paths, prior to performing a privilege restriction.

Using libraries can also complicate privilege restrictions. The libc library abstracts away complicated interactions with system calls. Developers attempting to perform privilege restrictions while making use of libraries would have to identify privilege uses within the library. This would require manually analyzing the library, creating an additional burden.

The Pledge framework is an active project and was recently renamed from tame to pledge. As the Pledge framework is under active development, software using it must also be reanalyzed and refactored if changes are made to the Pledge framework itself, increasing the manual effort required when using pledges.

Guarded Pledging In addition to monotonic privilege restriction, developers using the Pledge framework can perform *guarded privilege restrictions*. Guarded privilege restrictions require programs to find the preconditions for a privilege i.e. the conditions under which a privilege may be required along a path. Listing 2.2, shows the same example as earlier aside from making the calls to pledge conditional upon the flag ENABLE_WRITE. The program only writes to a file if that flag is set to true, therefore, the program only needs the wpath privilege if the flag is set. This presents an opportunity to eliminate unneeded privileges earlier using a check on ENABLE_WRITE earlier in the program. If true, we know that the program will open a file with write privileges (line 16) and would need the wpath privilege. Otherwise, the program would continue without writing and write privileges can be safely removed. Lines 9-13 demonstrate conditional restriction by checking whether ENABLE_WRITE is set. As the privilege restriction is guarded by a condition on ENABLE_WRITE, this is known as guarded privilege restriction. While this presents an opportunity to eliminate unneeded privileges earlier, it requires precise reasoning about the preconditions that decide privilege uses throughout the program.

Preconditions for privilege uses may be dependent on external inputs, which might not be available until later in the execution of the program. Only after the variable is live can it be used to perform a check. Developers must identify the precise inputs a privilege use is dependent on and use it to a perform a privilege restriction.

The possible paths taken by the program must also be considered, as branch conditions determine the privilege uses. A privilege use could possibly be guarded by several conditional statements and nested within several layers of function calls. The conditions used to perform the restriction must be extracted by the developer.

```
1  if (ENABLE_WRITE) {
2      pledge("stdio rpath wpath");
3  } else {
4      pledge("stdio rpath");
5  }
6  FILE* fp = fopen("input.txt", "r");
7  // Requires rpath
8  ...
9  if (ENABLE_WRITE) {
10     pledge("stdio wpath");
11 } else {
12     pledge("stdio");
13 }
14 ...
15 if (ENABLE_WRITE) {
16     FILE* fp2 = fopen("output.txt", "w");
17      // Requires wpath
18     fputs(data, fp2);
19     pledge("stdio");
20 }
21 printf("Printing Output: %s", data);
22  // Requires stdio
23 ...
```

Listing 2.2: Guarded Privilege Reduction

Reasoning about these factors over a combinatorial number of paths throughout the program can be difficult. Motivated by the difficulty in performing privilege reductions, we built a tool to relieve the programmer of the burden of correctly inferring the conditions associated with privilege uses and performing monotonic guarded privilege reductions.

Chapter 3

Inferring Guarded Privilege Restrictions

This chapter presents a formal overview of the technique we apply to infer guarded privilege restrictions. The following sections demonstrate how we model the inference of privilege requirements in a program. Using a small program (Listing 3.1) we describe how we can determine the privileges that will be required by working backwards through the program. Then, we demonstrate how our analysis infers the conditions under which privileges will be required. Finally, we demonstrate how the inferred conditions can be used to perform privilege restriction by strategically identifying points to insert conditional calls to pledge.

```
1  void work() {
2      //φ3: "write"
3    int data = input();
4      //φ6: "write" if data = 0 and ENABLE_LOG is true
5      //     or data > THRESHOLD
6    if (ENABLE_LOG) {
7      //φ7: "write" if data = 0 or data > THRESHOLD
8      logIfZero(data);
9    }
10     //φ11: "write" if data > THRESHOLD
11   if (data > THRESHOLD) {
12     //φ13: "write"
13     record(data)
14   }
15   //φ16: no privileges
16   return;
17 }
```

Listing 3.1: A privilege using function

The Principle of Least Privilege suggests that privileges should be removed as soon as they are not required. Determining the conditions under which a privilege will be required allows us to use the conditions to determine if the privilege will be used in the future. At runtime a check on these constraints can be used to conditionally remove privileges that are known to be unnecessary. Privileges in the Pledge framework map to groups of system calls, and restricting a privilege for one group does not affect the others. Hence, we extract the requirements and conditions related to each privilege independently. Later, we combine the results for each privilege to perform privilege restrictions. For clarity, we restrict our discussion to a hypothetical write privilege for this chapter. The write privilege is a contrived example, similar to the wpath privilege from the Pledge framework, with the only difference being that write will be checked when interacting with a file while wpath is checked earlier when opening a file.

The example in Listing 3.1 shows a simple function that uses the write privilege. The program receives an external input and stores it in a variable, data (line 3). The data variable is then used as an argument in a call to the logIfZero function (line 8). The function call is conditional on a global flag, ENABLE_LOG (line 6). The logIfZero function writes to a file if the argument supplied to it is set to 0. This requires the write privilege. line 11-14 is another conditional function call. The record function unconditionally requires the write privilege but is only called if data is greater than the THRESHOLD value. The program ends after the call to record. Throughout the function, there are several conditions that affect whether privileges are required.

By stepping backwards through the program, we can determine the conditions that must be satisfied if privileges are used later in the program. The return statement does not need privileges itself, and since there are no privilege using functions after it, the program does not require privileges at this point (line 15). At the previous statement, the call to record introduces the unconditional need for a privilege (line 12). The call is guarded by the if statement, which makes the privilege requirement conditional prior to line 10. Like the previous function, the call to logIfZero also introduces a privilege requirement. However, logIfZero only requires the write privilege when data is set to 0, which means write is required when data is equal to 0 *or* when data is greater than the THRESHOLD. Further up, the guard on logIfZero adds another condition on the privilege requirement. Prior to line 6, a privilege is required when data is 0 *and* along with it, ENABLE_LOG is true; or when data is greater than the THRESHOLD. The condition is simplified at line 3 when data is assigned an external input. At this point, since we cannot make assumptions about data, we assume that prior to line 3, write is unconditionally required. Thus, the conditions prior to lines 3, 6, 8, 11, and 13 are each conditions which will be true when write is required later on in an execution of the program, after each respective condition. We use the symbol ϕ_i to represent these conditions, where i is the line number for the statement after the condition. We use a novel *Precondition Inference* technique to find these conditions.

3.1 Precondition Inference

The privileges a program requires are determined by the function calls in a program. This information must be made available at earlier points to enable early privilege restrictions. The static analysis propagates this information backwards along with the necessary conditions for the privileges. Using the set of conditions ϕ available immediately after a statement i, a set of conditions ϕ_i can be inferred which will be satisfied in all cases where a privilege may be used in the future. We refer to the inferred conditions as *privilege preconditions*, while the conditions ϕ available immediately after a statement are called postconditions. Upon termination, ϕ is assumed to be false. Our analysis is guided by a few rules that follow from Hoare Logic[18] for privilege preconditions over different program statements.

Function Calls. Function calls may introduce a new privilege requirement. In the example on line 13, the record function unconditionally requires the use of a privilege. Since the privilege must be held before it, the inferred precondition is simply true.

$$\{\mathsf{true}\}\ \mathsf{record(data)}\ \{\mathsf{false}\} \tag{3.1}$$

$$\Pi_{\mathsf{record}} \triangleq \mathsf{true} \tag{3.2}$$

Equation 3.1 shows the call to record from line 13 in Listing 3.1. The post conditions are shown on the righthand side of the call statement while the preconditions are shown on the left. Since there are no privileges required after the call the postconditions are simply false. Equation 3.2 shows the specifications for the function as a boolean constraint which is unconditionally true making the preconditions inferred also unconditionally true.

$$\{\phi \vee \Pi_{\mathsf{logIfZero}}(\mathsf{arg})\}\ \mathsf{logIfZero(arg)}\ \{\phi\} \tag{3.3}$$

$$\Pi_{\mathsf{logIfZero}}(\mathsf{arg}) \triangleq \mathsf{arg} = 1 \tag{3.4}$$

Unlike the unconditional record function, the $\mathsf{logIfZero}$ function is conditional on its argument. Using the specifications (Equation 3.4), the precondition is inferred as a boolean constraint $\Pi_{\mathsf{logIfZero}}$ which is disjunctively added to the preconditions. The postconditions ϕ are not affected and are inferred as part of the preconditions to the statement. The static analysis infers the preconditions for both of these functions using the general rule:

$$\{\phi \vee \Pi_{function}(arg)\}function(arg)\{\phi\} \tag{3.5}$$

where $\Pi_{function}$ is the boolean constraint modeling the privilege requirements of the function.

Assignments. Assignments to variables with a postcondition ϕ follows a simple rule,

$$\{\phi[x/y]\}\ x = y\ \{\phi\}$$

$\phi[x/y]$ is used to denote the set of conditions ϕ, where every occurrence of the variable x in ϕ is replaced with y. Two variables pointing to the same location in memory during execution are said to be aliasing. Aliasing presents a challenge when attempting to infer preconditions and is addressed later (Section 4.1.2).

Inputs. The data variable receives an input on line 3. When inferring the preconditions for line 3, the static analysis is forced to approximate the constraints on data to true. Since the disjunct data > THRESHOLD must also become true, the entire disjunction becomes true.

$$\{\mathsf{true}\}\ \mathsf{data} = \mathsf{input}()\ \{\phi\}$$
$$\phi \triangleq (\mathsf{data} = 0) \wedge (\mathsf{ENABLE\_LOG} = \mathsf{true}) \vee (\mathsf{data} > \mathsf{THRESHOLD})$$

Since inputs cannot be reasoned about statically, the analysis follows the general rule in Equation 3.6 when inferring preconditions over inputs where all constraints ρ on the variable receiving an external input are conservatively replaced with true.

$$\{\phi[\rho_{var}/\mathsf{true}]\} var = \mathsf{input}()\ \{\phi\} \tag{3.6}$$

where ρ_{var} is a conjunct containing var.

Branches. There may be different privilege conditions associated with the different paths of a branch statement. After the branch on lines 11-13 in Listing 3.1, no privileges are required. The write privilege is introduced by the call to record within the branching path. Since the privilege use depends on the the branch condition being true, the preconditions inferred are $\phi_{11} \triangleq$ data > THRESHOLD.

$$\{\mathsf{data} > \mathsf{THRESHOLD}\}\ \mathsf{if}(\mathsf{data} > \mathsf{THRESHOLD})\ \mathsf{record}()\ \{\mathsf{false}\}$$

Similarly, for the statement on line 6, the postconditions are data > THRESHOLD while the privilege using function within it requires a privilege when data = 0. Using the branch condition and the function specifications, the inferred precondition is: $\phi_{11} \triangleq$ (ENABLE_LOG = true $\wedge$ data = 0) $\vee$ (data > THRESHOLD).

$$\{\phi_{11}\}\ \mathsf{if}(\mathsf{ENABLE\_LOG})\ \mathsf{logIfZero}()\ \{(\mathsf{data} > \mathsf{THRESHOLD})\}$$
$$\phi_{11} \triangleq (\mathsf{ENABLE\_LOG} = \mathsf{true} \wedge \mathsf{data} = 0) \vee (\mathsf{data} > \mathsf{THRESHOLD})$$

When performing precondition inference over branches, the branch condition ρ is added as a boolean conjunct to the privilege preconditions generated by the statements within the branch, $\phi_{internal}$. The postconditions ϕ of the branch are added as a disjunct. The general rule is shown in Equation 3.7

$$\{(\phi_{internal} \wedge \rho) \vee \phi\} \ \mathsf{if}(\rho) \ \phi_{internal} \ \{\phi\} \tag{3.7}$$

3.2 Weakest Sufficient vs Strongest Necessary Preconditions

Our analysis ensures that after performing privilege restrictions, the program has the necessary privileges to finish execution. Using an approximate set of conditions for privilege restrictions may result in a restriction that removes a necessary privilege, resulting in the program being terminated by the operating system later in its execution. Thus, while it is impossible to avoid approximations due to loops and runtime inputs[10], an analysis attempting to infer privilege preconditions must ensure that the program has the necessary privileges to finish execution. In the context of static analysis, this is known as *soundness*. While this could be trivially achieved by granting the program access to all privileges and never restricting privileges, it would be at odds to the Principle of Least Privilege. Preconditions may be computed as the strongest necessary preconditions or their dual, the weakest sufficient preconditions. We choose to compute the strongest necessary preconditions for privilege uses as it enables our analysis to be sound even when performing approximations over loops and external inputs[10]. Our justification of this follows:

```
1 void looper(int n, int data) {
2   for (int i = 0; i < n; i++) {
3       logIfZero(data);    // May need write privileges
4     }
5   }
6   return;
7 }
```

Listing 3.2: A privilege using function within a loop

Weakest Sufficient Preconditions. When inferring weakest sufficient preconditions, an approximate set of preconditions may be inferred that when used for privilege restriction will result in the program being underprivileged (holding fewer than necessary) in certain cases. Inferring the weakest sufficient preconditions for a privilege, computes the preconditions that guarantee the use of a privilege later in the program, i.e. the conditions are logically *sufficient* for a privilege to be used in the future. The program in Listing 3.2 shows a loop with a function call that requires a privilege. A static analysis attempting to analyze the

privilege preconditions prior to the loop would be forced to over-approximate. It would be valid for the analysis to compute an over-approximate set of weakest sufficient preconditions for line 2 as

$$\phi_{approx} \triangleq n > 10 \wedge \Pi_{\mathsf{logIfZero}}$$

If ϕ'_{approx} evaluates to true at runtime, the program will definitely need the write privilege. However, it does not cover the case where $0 < n < 10$, where the program will also require privileges. Over-approximations for weakest sufficient preconditions are in the direction of logical sufficiency, which yields a correct yet logically stronger set of preconditions. Thus, when computing the weakest sufficient preconditions for a privilege, a set of conditions may be inferred that guarantee the use of privilege but do not cover all the cases under which a privilege is needed.

Strongest Necessary Preconditions. Our analysis computes the dual to Weakest Sufficient Preconditions, i.e. the Strongest Necessary Preconditions. This allows the analysis to be sound when performing approximations for preconditions. The necessary preconditions for a program are the conditions which are satisfied in all cases when a privilege is used. Note that the use of a privilege is not guaranteed from these conditions being satisfied. It may so happen that the privilege is not required due to a later condition that the analysis had to approximate.

For the loop in Listing 3.2, the most over-approximate necessary privilege precondition is true which means that the privilege is unconditionally required. This preserves the requirement for a privilege. Conversely, when computing the weakest sufficient preconditions, the most over-approximate precondition is false, which would suggest that the privilege can be removed resulting in a privilege violation when $n > 0$. Our analysis is sound by design as the approximations made are towards logical necessity, which makes the preconditions weaker. This guarantees that the analysis does not incorrectly infer that a privilege may be removed, ensuring its soundness.

Line 3 in Listing 3.1 shows an external input to the data variable. Since the input is only available when the program executes the statement at runtime, we cannot statically reason about it. When performing precondition inference over the statement for privileges, the effects of the input must be conservatively modeled. Our analysis approximates the constraints over the variable and infers the preconditions as $\phi_3 \triangleq$ true, keeping the analysis sound. Approximating the weakest sufficient preconditions over inputs yields false (stronger) as the most approximate preconditions, while the strongest necessary preconditions can be approximated to be true (weaker). Like before, a false privilege precondition would suggest that privileges can be removed while true would preserve it.

Summarily, our analysis computes the Strongest Necessary Preconditions instead of the Weakest Sufficient Preconditions. Doing so guarantees that the analysis will not incorrectly infer that a privilege can be removed when dealing with inputs and approximations.

3.3 Performing Privilege Restrictions

The analysis infers the necessary privilege preconditions prior to each statement in the program, which can be used for privilege restrictions. Consider the preconditions inferred prior to every program statement in Listing 3.1. Conditional privilege restrictions can be made using any of the inferred privilege preconditions ϕ_i. For instance, a conditional restriction can be made prior to line 11 with ϕ_{11} as the condition.

```
1  void work() {
2      //φ3: "write"
3    int data = input();
4      //φ6: "write" if data = 0 and ENABLE_LOG is true
5      //     or data > THRESHOLD
6    if (ENABLE_LOG) {
7      logIfZero(data);
8    }
9    if (data > THRESHOLD) {
10     pledge("write");
11   } else {
12     pledge("");
13   }
14   if (data > THRESHOLD) {
15     record(data)
16   }
17   return;
18 }
```

Listing 3.3: Conditional restriction with inferred preconditions

Recall that a call to pledge is used to specify the privileges that the program must retain. On line 15 in Listing 3.3, the program calls the record function if the value of data is greater than THRESHOLD. If this condition does not hold the write privilege can be safely removed prior to line 14. Similar privilege restrictions can be made prior to each statement in the program. For a statement i with inferred necessary preconditions ϕ_i, ϕ_i will be satisfied if a privilege using function is invoked at a point after the statement i. Conversely, if ϕ_i is not satisfied the privilege will not be required and can be safely dropped.

3.3.1 Finding where to drop privileges

With the preconditions available at each point in the program, the analysis can insert privilege restrictions prior to each program statement. This is undesirable as it obfuscates the code with calls to pledge and also makes the program less efficient. Instead, our analysis identifies the points at which it would be useful to perform privilege restrictions.

Listing 3.4 is a straight-line function with conditional privilege restrictions inserted prior to every statement in the program. The statements from the input program prior to

```
1  int getNextPoint() { //*
2    pledge("write");
3    int data  = input(); //*
4    if (data == 0) {
5      pledge("write");
6    } else {
7      pledge("");
8    }
9    int pointA = data; //*
10   if (pointA == 0) {
11     pledge("write");
12   } else {
13     pledge("");
14   }
15   int pointB = generate(data); //*
16   if (pointA == 0) {
17     pledge("write");
18   } else {
19     pledge("");
20   }
21   logIfZero(pointA); //*
22   pledge("");
23   return pointB; //*
24 }
```

Listing 3.4: Redundant restrictions

```
1  int getNextPoint() {
2    pledge("write");
3    int data  = input();
4    if (data == 0) {
5      pledge("write");
6    } else {
7      pledge("");
8    }
9    int pointA = data;
10
11
12
13
14
15   int pointB = generate(data);
16
17
18
19
20
21   logIfZero(pointA);
22   pledge("");
23   return pointB;
24 }
```

Listing 3.5: Useful restrictions only

instrumentation is marked with a * as a comment beside the code. The privilege requirement is introduced on line 21. The argument deciding the need for a privilege is the pointA variable. On line 9, data is assigned to pointA. Earlier, data is initialized with an external input on line 3. The remaining statements in the program do not affect the pointA variable. Thus, while the privilege use is conditional on pointA, most of the statements in the program do not modify the privilege preconditions. Furthermore, since pointA is a reassignment of the external input, a conditional restriction on pointA can be replaced with a conditional restriction on data earlier in the function. Listing 3.5 shows a single conditional restriction after the input statement. The conditional restriction on line 4, using the data variable, removes the need for further restrictions till the final restriction on line 22 as the rest of the restrictions use the same condition.

Our analysis is able to eliminate these redundant restrictions by identifying the points at which the privilege preconditions are changed. Branches, function calls and external inputs *may* change necessary conditions for privilege uses later in the program. Other statements, such as assignments, *may* replace the variables or terms the privilege preconditions are dependent on but do not affect the underlying constraints. Since not all programs statements affect privilege preconditions, our analysis uses a novel single step forward analysis to find the points at which privilege restrictions are useful.

3.3.2 Identifying useful restrictions

According to the Principle of Least Privilege, privilege restrictions should be performed as early as possible. Using the privilege precondition $\phi_3 = \mathsf{true}$ prior to line 3 in Listing 3.1 however, unconditionally restricts the program to the write privilege. This means that the program will still hold on to the write privilege when ENABLE_LOG is false and the incoming value of data is less than THRESHOLD.

```
1  void work() {
2    int data = input();
3    if (data == 0 && ENABLE_LOG || data > THRESHOLD) {
4      pledge("write");
5    } else {
6      pledge("");
7    }
8    if (ENABLE_LOG) {
9      logIfZero(data);
10   }
11   if (data > THRESHOLD) {
12     record(data)
13   }
14   return;
15 }
```

Listing 3.6: Conditional restriction after input

Listing 3.6 shows a conditional restriction made after line 3 using the privilege preconditions ϕ_6. The condition is on the data variable using the external input only available after line 3. Intuitively, conditional privilege restrictions are more effective after the input variable that affects the privilege requirements is available.

Round-trip Inference. When the analysis infers privilege preconditions over an input statement, the constraints in the postcondition that are dependent on the input are conservatively made true in the preconditions. Such statements in the program affect the privilege preconditions inferred prior to it. Our analysis uses a novel round-trip analysis technique to identify such statements. A privilege restriction can then be inserted after such a statement.

The preconditions ϕ_i for a statement i are inferred from an available set of postconditions ϕ. Once ϕ_i is available for each statement, the analysis infers the strongest necessary postconditions ϕ^{post} for each statement that will be true after the statement executes. We refer to these conditions as the *inferred postconditions*. Equations 3.8 and 3.9 show the effects of a backward and then forward analysis over the input to data (line 3 in Listing 3.1) respectively.

$$\{\phi[\rho_{\mathsf{data}}/\mathsf{true}]\}\ \mathsf{data} = \mathsf{input}()\ \{\phi_6\} \tag{3.8}$$
$$\phi_6 \triangleq (\mathsf{data} = 0 \wedge \mathsf{ENABLE\_LOG} = \mathsf{true}) \vee (\mathsf{data} > \mathsf{THRESHOLD})$$

$$\{\phi[\rho_{\mathsf{data}}/\mathsf{true}]\}\ \mathsf{data} = \mathsf{input}()\ \{\phi^{post}\} \tag{3.9}$$

When inferring preconditions, the analysis replaces every conjunct on data in ϕ with true. Note that in this case, this results in the privilege precondition becoming true since all disjuncts contain a conjunct containing data. This means that the privilege is unconditionally required prior to the input. The inferred postcondition ϕ^{post} from $\phi[\rho_{\mathsf{data}}/\mathsf{true}]$ will also be simply true.

Examining the relationship between ϕ_6 and ϕ^{post}, we can see that $\phi_6 \implies \phi^{post}$ while $\phi^{post} \not\Longrightarrow \phi$. Intuitively, this identifies a point after which the conditions necessary for a privilege to be used in the future are changed. A privilege restriction at this point would use a stronger set of constraints to further reduce the conditions under which the program has access to the privilege. In the example, the analysis infers the input (available after line 3) as a point after which privileges should be conditionally restricted.

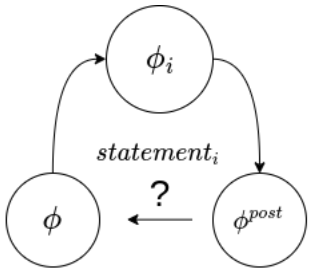


Figure 3.1: Visualizing the postcondition inference check

The analysis follows a general rule to find the points where privilege restrictions are useful. The analysis compares the *inferred postconditions* (ϕ^{post}) with the *available postconditions* (ϕ) to identify the points after which the conditions affecting the use of privileges are changed. When the inferred postconditions do not imply the available postconditions, the analysis interprets it as a point after which a privilege restriction should be inserted. Figure 3.1 visualizes the round-trip inference followed by the implication check for a single $statement_i$. Putting it all together, the preconditions ϕ_i are inferred from the postconditions ϕ. The postconditions ϕ^{post} is then inferred from ϕ_i. Finally, the analysis checks whether $\phi \impliedby \phi^{post}$.

3.4 Final Restrictions

Listing 3.7 shows the final version which has been instrumented with the useful privilege restrictions as inferred by our tool.

```
1  void work() {
2    int data = input();
3    if (data == 0 && ENABLE_LOG || data > THRESHOLD) {
4      pledge("write");
5    } else {
6      pledge("");
7    }
8    if (ENABLE_LOG) {
9      logIfZero(data);
10     if (data > THRESHOLD) {
11       pledge("write");
12     } else {
13       pledge("");
14     }
15   }
16   if (data > THRESHOLD) {
17     record(data);
18     pledge("");
19   }
20   return;
21 }
```

Listing 3.7: Final restrictions

The privilege requirement is introduced by the calls to logIfZero and record. By inferring the strongest necessary preconditions and then performing the round-trip analysis, the analysis identifies the point after the input as a useful point for lowering. The function calls are also identified as points to lower as the implication relationship between the inferred postconditions to the function call are different from the available postconditions. Note that using the rules presented in this chapter, a privilege restriction would also be inserted after line 15 as a consequence of modelling branches in the program as atomic (indivisible) units. In reality, PPIR is able to identify them as redundant as we implement our analysis with richer semantics, which we delve into in the next chapter.

Chapter Summary This chapter shows how we formalize the notion of conditional privilege restrictions which is required for inserting guarded pledges in a program. The foundations for our static analysis are laid out in this chapter and fleshed out fully with richer detail in the succeeding chapters.

Chapter 4

Static Analysis

Dataflow analyses[20] are a class of static analysis techniques employed by a compiler for optimizations, liveness analysis, dead code elimination and more. A dataflow analysis sets up constraints for each node in the control flow graph of a program. We define the constraints as flow equations over entry and exit states from neighbouring nodes, originating from a prespecified entry node. A state here refers to an abstract state, which we describe later. Repeatedly solving these equations yields a fixed point result that is an over-approximation of the necessary preconditions for privilege uses prior to each node.

4.1 Intra-procedural

We describe the intra-procedural analysis using the three components of a monotone dataflow analysis[27] framework: the abstract state and domain, the transfer functions, and the join operator with the help of a small, kernel language (similar to LLVM IR). Later, we show the *inter-procedural* effects of the algorithm to show how we can analyze whole programs.

While relatively simple, the kernel language described by the grammar in Figure 4.1 is enough to model the complexities typical to real world programs. A program in this language is built from one or more functions. Each function consists of a set of statements and always ends with a statement returning a value, the return statement. Control flow is modeled through conditional and non-conditional goto statements and function calls. The language also models interaction with memory via loads and stores. We use this to describe our approach to handling aliases in real world programs in Section 3.2.2. We abstract away the details of exact mathematical operations and represent them as program expressions (*exprs*) val op val and op val for unary arithmetic respectively. *Exprs* in our language return values from computations, functions calls or return addresses of variables. Note that, while the kernel language limits the use of pointer arithmetic, our implementation of the analysis has no such limitation. Finally, we model the use of programming constructs that cannot be reasoned about as *havoc* expressions.

$$
\begin{array}{lll}
\textit{stmt} & ::= & \\
 & | & \textit{var} = \textit{expr} \\
 & | & \texttt{ptr} = \textit{expr} \\
 & | & \texttt{store}\ \textit{val}, \texttt{ptr} \\
 & | & \texttt{if}\ \mathbf{p}\ \texttt{goto}\ l' \\
 & & \\
\textit{expr} & ::= & \\
 & | & \textit{var}\ \textit{op}\ \textit{var} \\
 & | & \textit{op}\ \textit{var} \\
 & | & \texttt{addr} \\
 & | & \texttt{load}\ \texttt{ptr} \\
 & | & \textit{havoc} \\
 & & \\
\textit{var} & ::= & \\
 & | & \textit{val} \\
 & | & \texttt{ptr}
\end{array}
$$

Figure 4.1: The kernel grammar used to describe our analysis

4.1.1 Abstract State and Domain

We model an abstract state as a predicate ϕ, where for a pair of abstract states, ϕ_A and ϕ_B, $\phi_A \preceq \phi_B$ *if* $\phi_B \implies \phi_A$. The partial ordering between abstract states forms an abstract lattice or abstract domain. The top of the lattice is set as *true* (logical truth, $\top$) and conversely, the bottom as *false* (logical falsehood, $\bot$), naturally following the partial ordering. A merge between two abstract states is defined by a join operation $\phi_A \sqcup \phi_B$, which results in a move up the lattice to the least upper bound of ϕ_A and ϕ_B, $\phi_{ub} \sqsubseteq \{\phi_A, \phi_B\}$. During the analysis, approximations can be made by forcing the abstract state to $\top$.

Each abstract state is a disjunction, $\delta_1 \vee \delta_2 \ldots \delta_n$ where δ_i is a conjunction (alternatively, a disjunct) of program constraints, $\rho_1 \wedge \rho_2 \wedge \ldots \rho_n$. Modeling the abstract state disjunctively (i.e. in DNF form) enables the analysis to capture the privilege preconditions that arise from different branches in the program (Section 3.1). As demonstrated by the motivating example (Listing 2.1), a privilege use may be dependent on a constraint, the constituents of which may be modified through the course of the program. A ρ in our abstract state models a program constraint as a logical formula of program instructions that is updated through transfer functions as the analysis progresses through the program.

To represent the complete set of privileges we use a tuple of abstract states. Each element in the tuple is associated with a single privilege. This does not introduce additional complexity to the algorithm as privileges are independent of each other and, transitively, so are the abstract states for each privilege. Considering the independence of privileges, we present the analysis from the perspective of a single privilege.

	Instruction i	λ**TransferFunction.**$i.\phi$
ASSIGNMENT	*var* = *expr*	$\phi[var/expr]$
HAVOC	*var* = *havoc*	$\rho_{var} = \top, \forall \rho_{var} \in \phi$
BRANCH	if ρ goto l	$\delta = \delta \wedge \rho,\ \forall \delta \in \phi$
	store var, ptr_s $\exists$load $\text{ptr}_l \in \phi$: MustAlias(ptr_s, ptr_l)	ϕ[load ptr_l/var]
STORE	store var, ptr_s $\exists$load $\text{ptr}_l \in \phi$: MayAlias(ptr_s, ptr_l)	$\delta = \delta$[load ptr_l/var] $\wedge$ (ptr_s = ptr_l), $\forall$ load $\text{ptr}_l \in \phi$
	store var, ptr_s $\nexists$load $\text{ptr}_l \in \phi$: MayAlias(ptr_s, ptr_l) $\vee$ MustAlias(ptr_s, ptr_l)	skip

Table 4.1: Transfer Functions

4.1.2 Transfer Functions and Aliasing

A transfer function $\lambda Transfer(stmt, \phi)$, infers the preconditions for a program statement *stmt*, given the statement itself and an abstract state ϕ as a set of postconditions. An application of $\lambda Transfer$ returns a new abstract state ϕ' that are the preconditions for *stmt*. The transfer functions follow the rules laid out in Table 4.1.

For an assignment, every occurrence of *var*, which may either be a variable or a pointer, in ϕ is replaced with the *expr* that's been assigned to it. The assignment rule captures arithmetic operations, values loaded from pointers, and addresses of variables as the language represents each as an *expr*.

The transfer functions applied over load and store expressions are far more interesting. A language that admits the use of loads and stores to memory, also introduces situations where two pointers may address or alias the same location in memory. From a static analysis perspective, this presents a challenging problem[8][23]. We discuss our approach to dealing with aliases in the latter part of this section.

```
1 func foo():
2   ...
3   x = havoc
4   ...
5   file = open(x)
6   ...
7   return 0
```

Listing 4.1: Straight Line Program Using Conditional Privileges

For the straight line program in Listing 4.1, the use of a privilege is determined by the input to the program stored in the variable x. Without any specifications constraining the input, the value of x prior to the input is impossible to know. We model such an event in our kernel language as $var = \mathsf{havoc}$. Per the rules of our analysis, given a ϕ, the transfer function operating over a havoc expression sets ρ_{var}, i.e. every conjunct that contains var in ϕ to true. Replacing conjuncts to true in a disjunction ϕ yields a ϕ' such that, $\phi' \implies \phi$. This preserves the ordering of ϕ and ϕ' in the lattice where ϕ', being weaker, is higher up the lattice.

We model all non-invertible functions, i.e. a function whose arguments cannot be inferred from its results, as a havoc assignment. Technically, non-invertible but idempotent functions, such as arithmetic operations can be modeled as an element in ρ. Consider the program statement $\mathsf{y} = \mathsf{x}\ \mathit{mod}\ 3$, which performs a modulus operation on a variable x. Given the postcondition to the modulus operation as $\rho \triangleq y = 2$, inferring the exact value of x prior to that line is impossible. On the other hand we can absorb the modulus operation into our logical formula as $\rho' \triangleq \mathsf{x}\ \mathit{mod}\ 3 = 2$. This is possible due to the fact that a modulus operation is idempotent, i.e. the modulus operation will produce the same results given the same inputs.

While the analysis does take advantage of idempotency in arithmetic operations, identifying idempotency over several lines of code is a challenge which we do not attempt to tackle. Other programming constructs that are non-invertible include hash functions, loops, and external functions for which specifications aren't available.

Aliasing. Aliasing refers to the situation where two variables point to the same location in memory, which makes precise static analysis of programs difficult. For the purposes of our analysis, aliasing presents a challenge as privilege uses may be dependent on pointers which may or may not alias. As a result updates through aliasing pointers must be captured by the abstract states. Programming languages permit the use of aliasing through the use of pointers and reference variables. The kernel grammar prevents the use of reference variables but allows the use pointers since reference variables, while presented differently to a programmer, are equivalent to pointers under the hood of a compiler.

We use three helper functions, MustAlias, MayAlias and NoAlias, to determine the aliasing relationship between two pointers. Two pointers *must alias* if it can be proven that they point to the same location in memory. Similarly, a *no alias* relationship exists when it can be proven that they do not point to the same location. These relationships require strong proof since taking advantage of either of them when unproven, can lead to memory corruption errors. When neither can be proven, the pointers are said to have a *may alias* relationship. Alias analysis is a well studied problem for which various solutions exist[38][28], and we resort to them for determining the aliasing relationships between pointers. Additionally, the kernel grammar restricts the use of pointers to load and store statements exclusively.

```
1 func foo(ptr py):
2   val x = 5
3   ptr px = &x
4   store 0, px
5   val flag = load px
6   ReadWrite("Aliasing is Hard", px);
7   return 0
```

Listing 4.2: Conditional Privilege Requirements due to Aliasing

Listing 4.2 is a contrived example demonstrating the use of privileges dependent on pointers. The call to the function ReadWrite only requires a privilege if the second argument to the function, flag has an integer value of 2. flag depends on the value assigned to it by the load statement in the previous line. Consider that the pointer argument px to the load instruction may alias the pointer py. Pointer py has a new value stored to it at line 4, the result of the expression from line 3. For the analysis to be sound, it must be able to capture the semantics of the cases where px and py must, may and may not alias. The transfer functions capture aliasing behaviours of a program through operations on store instructions. A load instruction is treated as an assignment expression since at runtime when the program reaches a load, the argument to the load will be unambiguous.

For a store instruction store $\mathsf{var_s}$, ptr_s, the transfer function checks the pointer argument ptr_l to every load instruction load currently in the abstract state for its aliasing relationship with ptr_s. If ptr_l must-alias, we consider it an assignment; the use of the load instruction is simply rewritten with $\mathsf{var_s}$. A no-aliasing relationship results in no change to the abstract state.

In the event that a ptr_l may-alias ptr_s, each disjunct δ containing ptr_s is duplicated into the abstract state with an additional conjunct $ptr_s \neq ptr_l$, i.e. ptr_s does not alias ptr_l. At the same time, every conjunct ρ_l in δ that held a use of the of ptr_l is rewritten with var_s as $\rho'_l[load/var_\mathsf{s}]$ and an additional conjunct $ptr_s = ptr_l$, is appended to δ. Intuitively this represents the two outcomes of a may aliasing relationship, it either must or it must not.

An abstract state can have several load instructions, consequently, the pointer associated with each can have differing relations with the store pointer being analyzed. To deal with this, aliasing is dealt with in three passes for each store instruction using the rules mentioned for each kind of relationship. For a set of load instructions and a store pointer, the first deals with no-aliases, the second with must aliases and the third with may-aliases. The first two passes reduce the load instructions to be operated on, and the third conservatively transforms the remaining.

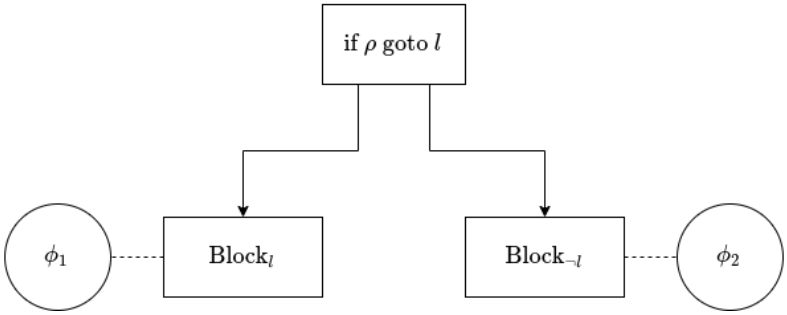


Figure 4.2: Control Flow Example

4.1.3 Edge Transformations and Join Operator

Control flow structures in a program demonstrate the path sensitive nature of privilege uses. Consider a program with three branches, each guarded by an independent conditional statement. The first and second branches exit after calling a function that requires a privilege. The program will need the privilege when either the first *or* the second guard condition is true. As such, the abstract state models the disjunctive program constraints introduced by conditional statements. The kernel language restricts the control flow to binary branching. This is enough to model branching structures such as switches and if-else ladders. A conditional statement in a program written in the kernel language takes the form if ρ: goto l. The boolean variable ρ, determines whether the program will jump to block$_l$, the program block with the label l. We model ρ as a conjunct in our abstract state.

Prior to the evaluation of a conditional statement, a program potentially requires the privileges on either branch of the control flow. Inside block$_l$, the preconditions for the use of the privilege is independent of the guarding condition since it *must* be true at that point. Similarly, the preconditions for a privilege use within the program block associated with the false edge of the conditional statement are also independent of the guarding condition since it must have been false. The backwards analysis models this by first considering the function call using a privilege to be unconditional, save the constraints that are inherent to the function (flag variables etc.). Later, as the analysis proceeds backwards over the conditional

	Instruction i	λ**Transformer.**$i.\phi$
Edge Transform	if ρ goto l	$\phi' \triangleq \{\forall \delta \in \phi_\rho, \delta = \delta \wedge \rho\}$ where $\phi_\rho \in Edge_{i,l}$
	if $\neg\rho$ goto l'	$\phi' \triangleq \{\forall \delta \in \phi_{\neg\rho}, \delta = \delta \wedge \neg\rho\}$ where $\phi_{\neg\rho} \in Edge_{i,l'}$
Join Operator	$\phi_1 \sqcup \phi_2$	$\phi' = \phi_1 \vee \phi_2$

Figure 4.3: Edge Transformers and Join Operator

statement, it attaches the conjunct ρ in its straight and negated form, to the abstract states associated with the false and true edges respectively. To maintain the disjunctive form of the abstract state, conjuncts inferred by control flow structures are inserted into every disjunct in an abstract state. This is followed by merging the abstract states with a join operation.

A backwards merge in the control flow necessitates a merge over abstract states, defined by the join operator $\sqcup$ in a dataflow analysis. The path sensitive requirements of the analysis require that the abstract states correctly represent the privilege requirements for each path. We define the join over two abstract states ϕ_1 and ϕ_1 as $\phi_1 \sqcup \phi_1 = \phi_1 \vee \phi_2$; $\sqcup$ performs a logical *or* operation over the two abstract states resulting in a disjunction of program constraints, ϕ', associated to either side of a conditional statement.

Algorithm 1: Precondition Inference Algorithm

input : Function

```
1  exits ← IdentifyExits(Function)
2  worklist.add(exits)
3  state ← map : BasicBlock to φ // map of blocks to abstract states
4  while worklist is not empty: do
5      block ← worklist.deque()
6      for s in Successors(block) do
           // Apply edge transformations to each successor
           state[block] ← transfer(edge(block, s), state [s]) ⊔ state[block]
7      for inst in reverse(block) do
           // Apply transfer function on each instruction in the basic block
           state[block] ← state[block] ⊔ transfer(inst)
8      for p in Predeccessors(block) do
9          if p not in worklist then
10             worklist.add(p)
```

output: Preconditions for each instruction as a map

Algorithm 1 shows the pseudocode for the algorithm used to implement the static analysis for precondition inference. The algorithm accepts a function as an input and identifies the exits which are stored in a worklist. A basic block is taken from the worklist and is processed backwards using the edge transforms, transfer functions and join operator. The output for the algorithm is a map of statements to their preconditions.

4.2 Inter-procedural

Privilege restriction applies over the whole program. Consequently, the analysis must be able extract conditional privilege uses inter-procedurally. Functions may also be called multiple times at multiple places indicating the need for context sensitivity. Our analysis is implemented within a *k-CFA* framework[36] with $k = 2$.

A function call may be of 3 types, internal, external and indirect. An internal call refers to calls made to functions that exist within the same program. External calls are calls to functions which are only declared within the program but not defined. Finally, indirect calls are made through function pointers. While it is possible to make external calls through indirect calls, it's rare in practice and we do not account for it. Many programs are built using libraries and for the analysis to be successful, the input program to the analysis must also contain the libraries as part of the same module.

Internal Calls When the analysis encounters a call to a function for which the definition is available, the analysis halts progress in the caller function and begins analyzing the called function. We use a simple call strings based approach for context sensitivity. A call string is a stack of call locations, used to identify the context (where the function was called from) in which a function is being analyzed[35].

Indirect Calls Indirect calls are handled similar to internal calls. We rely on an external alias analysis to identify the targets of the indirect call. Each target is analyzed independently with a copy of the abstract state being sent into each, rewritten with the argument of the return statement. Once the analysis of the targets are finished, the results are merged together through a join operation. The analysis of the caller function proceeds using the merged results.

External Calls As external calls cannot be analyzed we consider them to be *havoc* statements and conjuncts using those values are pushed to true. Additionally, if the call requires the use of a privilege, the abstract state corresponding to the privilege is set to true.

Algorithm 2: Lowering Location Algorithm

input : Translation Unit (Module), Preconditions

```
1 for inst ← in Module do
2     post ← transfer(preconditions[inst], inst)
3     if post ⟹ preconditions[next(inst)] then
          // inferred postconditions ⟹ available postconditions
4         continue
5     else
6         locations.add(inst)
```

output: List of instructions

4.3 Forward Inference

With the preconditions available for each instruction in the program the analysis uses Algorithm 2 to find the points in the program where it should insert conditional privilege

restrictions using the postcondition inference idea illustrated in Section 3.3.2. For each instruction in the program, the analysis infers the postconditions to the instruction using the preconditions inferred from Algorithm 1. Then the analysis checks if the postconditions inferred imply the preconditions of the next instruction. When the implication relationship does not hold, the analysis identifies that location as a point to insert a privilege restriction. Otherwise, the analysis continues to the next instruction, discarding the inferred postconditions.

4.4 Summary

This chapter described our approach to building the ideas described in the previous chapter as a static analysis as a Dataflow Analysis. The join operators, transfer functions and abstract state for the Dataflow Analysis were illustrated with the help of a small kernel language. This chapter also presented the algorithms to be implemented and how to extend it to be inter-procedural.

Chapter 5

Implementation

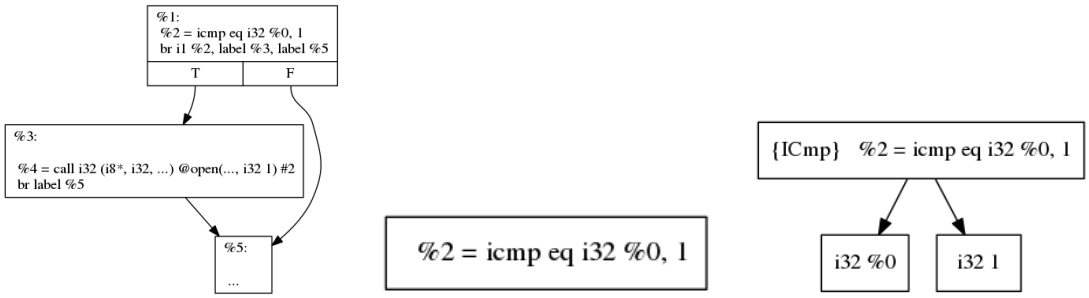
The following sections describe how the abstract state for the analysis is implemented with regard to the data structures used and the required simplifications. This chapter also includes a brief introduction to the LLVM compiler framework[24] which describes the necessary details to understand our implementation.

5.1 Constraint Trees

We model the constraints in disjunctive normal form (DNF). Each conjunct in the disjunctive sets, or disjunction, is built as a binary tree, similar to abstract syntax trees, which we call a constraint tree. The tree representation allows easy modifications when performing rewrites through the transfer functions (Section 4.1.2). The trees are maintained as proper binary trees, i.e. each node will have either 0 or 2 children, and each node is assigned a unique identifier. When a constraint is picked up from the program, it is introduced as a leaf node and assigned a new identifier. Figure 5.1a shows an example where a constraint is introduced from the conditional use of a privilege determined by a branch statement. The value of the branch condition is introduced to the system as a leaf node (Figure 5.1b), forming the root of a constraint tree along with the form of the constraint, false. As the analysis proceeds backwards up the program and reaches the instruction which produces the value of the branch condition, the icmp instruction, the leaf node is rewritten as a binary node (Figure 5.1c) with two leaf nodes of the form, $\mathsf{inst}_{\mathsf{lhs}}$ *op* $\mathsf{inst}_{\mathsf{rhs}}$ where *op* is a comparison operator which produces a boolean value. While the right child node is a constant value, the left node is a value produced during the execution of the program. Similar to the update made to the branch condition, the left node will also be rewritten when the analysis reaches its definition.

5.2 Simplifications

Each node in the binary tree, including the root node, has a unique identifier with which we can identify each constraint in the set and apply logic based simplification rules after



(a) Guarded Privilege Use (b) Constraint Tree Root (c) Final Constraint Tree

Figure 5.1: Constraint Tree Formation

every join and transfer operation. Simplification eliminates redundant constraints from the analysis, which enables the analysis to scale[10]. Our objective is to compute a minimal yet equivalent disjunctive set of constraints that characterizes the need for a privilege at any point in the program. A minimal set of constraints primarily enables easier analysis of the constraints and also reduces the time taken to analyze a program. Due to the path sensitive nature of the analysis, without effective simplification, the constraints grow to the order of thousands, which bogs down the analysis.

Switches. We find switches to be difficult to deal with as they introduce a large number of constraints. A boolean based simplifier is unable to simplify these constraints as the edge transformation introduces a new conjunct into each abstract state associated with each successor. Instead, we apply a simple rule to collect the intersecting disjuncts $D_{instersect} = \forall i \bigcap D_i$ across each incoming disjunction D_i. $D_{instersect}$ represents the set of constraints that are unaffected by which branch of the switch the program executes. The resulting disjunction at the switch then becomes $\phi' = D_{intersect} \cup \phi_i$. Where ϕ_i is the incoming disjunction, D_{rem} = D_i / $D_{instersect}$ from the i^{th} successor to the current switch as transformed by the edge transform operator. This operation generalizes to if-else branches as well as indirect calls that have more than one target.

Constant Evaluations. Another opportunity for simplification arises when the leaves of constraint tree are constants. As depicted in Listing 5.2, suppose a program sets the value of a variable to different constant values depending on the branch the program executes. On line 8, the value of the variable is checked in a guard before calling a privilege inducing function that is picked up by the analysis as a constraint tree. When it reaches the constant assignment on each of the paths i.e. lines 3 and 5, the guard variable g can be rewritten with the constant value being assigned to it. On each of the paths, the constraint tree then represents an operation on constant values which can be folded to produce a boolean result using LLVM's Constant Folder. Since the constraints trees are maintained in disjunctive normal form, when a conjunct is evaluated to true, it is removed from a disjunct. When false, the disjunct the conjunct belongs to becomes false and is removed from the disjunction.

```
int foo() {
  ...
  int write_to_file = 0;
  if (g > 2) {
    write_to_file = 1;
  }
  ...
  if (write_to_file == 1) {
    fopen(...);
    fprintf(...)
  }
  ...
```

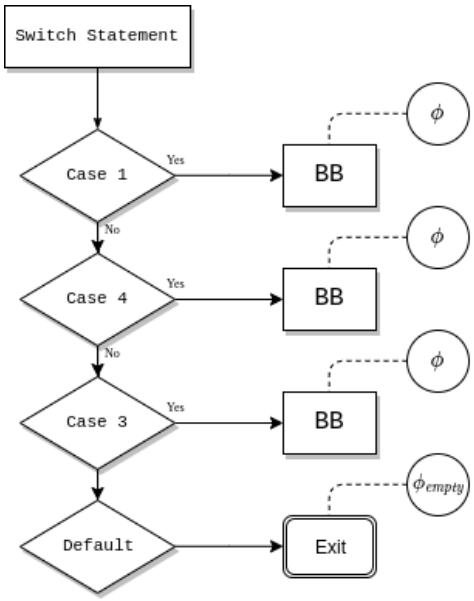
Listing 5.1: Program Fragment with Constant Evaluable Constraints

5.3 Havocs and Approximations

A *havoc* expression represents a point in the program that the analysis cannot reason about. When the analysis encounters a *havoc* expression, any constraint tree that uses the value dependent on a *havoc* expression is set to true. *havoc* expressions also allow a unified way of modelling approximations by considering certain constructs in the input program to be *havoc* expressions and handling them as such. Table 5.1 lists the approximations we make. Approximations for arrays and recursive data structures were necessary to make analysis scale to even the smallest of programs[25]. We find that field accesses caused difficulty in performing the analysis and we approximate them when they have may-aliasing relationships with other pointers in the constraint set or when the field holds a pointer type.

Exit Paths. Exit paths add significant complexity to programs. When evaluating the conditions for a switch, a malformed input may cause the program to pick the exit path. Often the exit path requires no privileges, while the other paths have common disjunctive constraints under which privileges are required. The optimization we use at branches is ineffective since the intersecting disjuncts becomes the empty set. Each incoming disjunction is retained disjunctively and the abstract state quickly grows to the point where the analysis is unable to proceed. To get around this, we do not consider empty disjunctions from exit paths when performing a join over backwards merges in the control flow, i.e. $D_{instersect} = \bigsqcup Successors()$, where *Successors* is a function that returns abstract states for the successors of the current states except the empty states from exit paths. A similar technique is also used by Aracde, where the error paths taken by the program are removed from the abstracted graph.

Figure 5.2a shows an example switch statement where the default case leads to an exit. Assume that the non-default cases coalesce into a single path before any privilege using



(a) Exit Path Simplification

$$D_{intersect} = \phi_1 \vee \phi_2 \vee \phi_3 \tag{5.1}$$

(b) Preconditions without approximation

$$D_{approx} = \phi \tag{5.2}$$

(c) Preconditions with approximation

system calls are invoked. The incoming abstract state ϕ is then common to all the non-default cases and is indicated with a dashed line while the incoming abstract state from the default case (ϕ_{empty}) is empty. Without the approximations the join operator would attach conjuncts to each incoming abstract state to represent the associated case. This creates three new disjuncts (Equation 5.2b). Ignoring the empty abstract state from the exit path allows the common disjunction to be inferred as the precondition across all three cases (Equation 5.2c), simplifying the analysis while maintaining soundness.

Non empty states arising from exit paths are treated as regular paths. From a security perspective, this eliminates the possibility of performing privilege restrictions prior to a program executing an exit path. Since the program would exit along the path anyway, we believe that this is not a great concern.

Loops. Loop approximations are a necessity for a static analysis due to the halting problem. For the purposes of our analysis we perform a simple approximation that pushes the constraints that are introduced or modified within the body of a loop to true when they are propagated along the backedge of the loop to the header. The loop header then retains the set of constraints that are not modified by the loop body and the constraints that are introduced in the first backward iteration of the loop. This is similar to the methodology used by Arcade[1], where the program's control flow graph is abstracted to a directed acyclic graph.

Approximation on False Constants. As described previously, we can perform constant folding operations on our constraint trees to produce boolean results. A false result leads to

Program Construct	IR Description
Array Accesses	Values accessed through GEPs into arrays (including global and constant valued arrays)
Recursive Data Structures	Values accessed through GEPs or Loads which have the same base type as the pointer operand being indexed into
May-Aliasing Field Accesses	Store value operands which use loads from GEPs
Interprocedural May-Aliases	May aliasing operations for load/store pairs which occur in different functions
May-Aliasing Double Indirection	Store value operands which are from Loads and are pointers

Table 5.1: Approximations

the disjunct being removed. We choose this as a point of approximation by pushing up the lattice and making the disjunct vacuously true, which makes the abstract state vacuously true at that point. Pushing the abstract state to true at that point forces the analysis to associate a privilege use along the path being analyzed which reduces the possibility of performing conditional privilege restrictions. Our lowering strategy, which we discuss in the next section, mitigates some of the loss in precision made by this approximation.

5.4 Lowering

The final part of the analysis involves inserting the conditions extracted by the analysis back into the program to be evaluated at run-time for restricting privileges. Remember that a call to pledge() should pass the list of privileges that should be held on to by the program. Given a set of constraints associated with each privilege, the lowering phase instruments the program with the conditions that when evaluated at runtime generate a call to pledge() with a string of privileges that should be retained by the program.

We select the points in the program to lower at by tracking the points where the analysis made approximations or encountered havoc statements. The constraints associated with each of these points are then merged over all contexts and lowered into the program. The constraints used to lower are also the constraints available prior to the approximations. This mitigates some of the loss in precision from the approximations mentioned earlier.

5.5 Specifications

Functions that require privileges must be identified by the analysis and to that end their specifications must be provided to the analysis externally. The man pages from OpenBSD

Privilege	Handler	Description
stdio		Permits system calls which interact with file descriptors including stdin and stdout along with a host of other non-side effecting system calls; also required by exit handlers
rpath	✓	Permits system calls to open files for reading
wpath	✓	Permits system calls to open files for writing
cpath	✓	Permits system calls to create files or directories
tmppath	✓	Permits system calls which make and modify files in /tmp directory. Redundant in the presence of rpath, cpath, wpath
inet	✓	Permits system calls to open and modify sockets in the AF_INET and AF_INET domains
fattr		Permits system calls which can make changes to file attributes
dns		Permits system calls to DNS network transactions
unix	✓	Permits system calls to open and modify sockets in the AF_INET domain
tape	✓	Permits the ioctl sytem call to operate on tape drives

Table 5.2: Privilege Specifications

provides a specification for system calls and the privileges they require. Out of the 31 privileges specified on the man-page, we support 10 of them as listed in Table 5.2.

The column 'Handler' in Table 5.2 shows whether a special handler is required for that privilege. As of now, the specifications for pledge do not adequately list the effects of arguments of system calls on the privileges required. For example, the open() system call can open a file for reading, writing or create one if it's not already present. These operations the rpath, wpath and cpath privileges respectively. Furthermore, the write system call does **not** in fact, require the wpath privilege. We see this idiom replicated across other system calls and build custom handlers for them. These handlers check the arguments to the system call and reports the correct set of privileges required by the system call.
Libraries such as libC also require privileges since they use system calls as well. As it would not be practical to reanalyze a library for each program its used in, we built a tool to perform a transitive closure over the library APIs and the system calls invoked subsequently to generate the privilege requirements for the APIs.

5.6 Summary

The dataflow analysis generates gathers the preconditions for privilege using system calls as boolean formulae. The abstract state for the dataflow analysis is built from disjunctive conjunctions of boolean formulae i.e. in Disjunctive Normal Form. These formulae are stored as

binary trees to enable easy term rewriting. Crucial to analyzing programs is simplification over branches, switches and constant terms in the program. Additionally, we also perform significant approximations for loops, exit paths and false constraints to help scale the analysis. With the preconditions for privilege uses available at each instruction, we lower them as conditions for guarded calls to pledge to perform privilege restrictions.

Chapter 6

Evaluation

We evaluated PPIR on 4 programs from the OpenBSD core utilities: cat, cp, dd and date. With the results from our evaluations we seek to resolve three concerns:

- Establishing the functional correctness of programs that have been instrumented to perform conditional privilege restrictions. (Section 6.1)
- Quantifying the reduction in attackable surface area with respect to manually instrumented programs. (Section 6.2)

Additionally, we seek to understand the nature of the conditions used when performing the conditional restrictions, the effect of the approximations we utilize, and the threats to the validity of our results.

6.1 Correctness

PPIR extracts conditions from a program, instrumenting it at various points to perform conditional privilege restrictions without inducing behavioural differences or underprivileging. A program attempting to invoke a system call without the correct set of privileges will be terminated by the operating system.

When using the Pledge framework, a program cannot regain privileges it has already given up. Establishing functional correctness is our first priority when testing the privilege restrictions made by PPIR. To ensure that the programs we instrument are not underprivileged we test their functional correctness using a test suite. Underprivileged programs will be terminated by the operating system and emit a log message indicating the required privilege.

Benchmark	Test Suite Provider	#Tests
date	BusyBox	1
cat	BusyBox	2
cp	BusyBox	11
dd	BusyBox	4

Table 6.1: Tests run for each benchmark

Correctly instrumented programs without privilege errors will pass the tests without logging an error. As the Pledge framework is a part of the of the OpenBSD operating system we have to test the programs on the OpenBSD platform. However, we were unable to find OpenBSD specific test suites for our benchmarks. Instead, we use the test suite from BusyBox, a distribution of POSIX compliant Linux utilities. Since our benchmarks are from OpenBSD and the test suite is built for BusyBox, some tests will fail due to behavioural differences of the OpenBSD and BusyBox implementations of the programs. These differences could lead to false detection of behavioural differences that are not due to PPIR. To prevent this, we first filter the tests that will cause such problems. We identify the tests that expect behaviour specific to the BusyBox implementations by first running the OpenBSD version of the program against the test suite. These programs have calls to pledge inserted in them by the authors. The failing test cases test for features specific to the BusyBox version of the program. We remove them and use the remainder of the test suite for evaluations. Table 6.1 shows the number of tests available per benchmark after removing the failing tests.

With the spurious test cases removed, we can test for behavioural differences between the instrumented and original benchmarks. Correctly instrumented programs will not exhibit any differences in behaviour from the original program or terminate early due to underprivileging. Since the test case checks the functionality of the program, a correctly instrumented program will pass all the test cases which were passed by the original program. The programs instrumented by PPIR passed all test cases that were passed by the original program. This shows that instrumented programs were not underprivileged and that the instrumentation does not change the behaviour of the program.

6.2 Reduction In Surface Area of Attack

A vulnerability at any point in the program may be exploited by an attacker, after which we consider the process to be controlled by the attacker. Following the Principle of Least Privilege, the Pledge framework can be used to limit the potential damage. With privilege restrictions in place, if the process were to be hijacked by an attacker, it would be terminated when the attacker attempts to perform an action that requires privileges. We conservatively assume every instruction in the program to be a location where privileges could be exploited.

With respect to the Principle of Least Privilege, our objective is to reduce privileges as early as possible, reducing the attack surface of the program. PPIR inserts conditional privilege restrictions in a program to enable privilege reduction at runtime. We compare the effectiveness of privilege restrictions in the programs instrumented by PPIR to the effectiveness of the privilege restrictions performed with expert knowledge by the authors of a program.

We can quantify the effectiveness by considering the number of privileges available to each instruction during the execution of a program[12]. When the program begins execution it will have access to all privileges. A call to pledge will restrict all subsequent instructions executed to a subset of the available privileges. Each instruction prior to a call to pledge executes with the full set of privileges available to it. The instructions executed after the call will have the subset of privileges specified in the call to pledge available to it. Each instruction executed can be associated with the subset of privileges that were held by it. Counting the executed instructions associated with each privilege indicates how early the privileges were dropped in the dynamic trace. The earlier the privileges are dropped, the fewer instructions associated with each privilege there will be. Earlier privilege drops being the preference, the version that has fewer instructions executed per privilege will be the preferred version.

Table 6.2: Active Instructions per Privilege

Privilege	Test 1		Test 2	
	Manual	PPIR	Manual	PPIR
all	2	1	2	1
stdio	166	171	149	153
rpath	166	165	149	147

Table 6.3: For cat

Privilege	Test 1	
	Manual	PPIR
all	49	0
settime	0	55
stdio	0	55

Table 6.4: For date

Privilege	Test 6		Test 7		Test 9		Test 10	
	Manual	PPIR	Manual	PPIR	Manual	PPIR	Manual	PPIR
all	6	0	6	0	6	0	6	0
cpath	383	550	403	570	332	502	403	570
rpath	383	550	403	570	332	502	403	570
stdio	481	550	501	570	430	502	501	570
wpath	383	550	403	570	332	502	403	570
tape	0	550	0	570	0	502	0	570
error	383	0	403	0	332	0	403	0

Table 6.5: Active Instructions per Privilege for dd

Privilege	Test 1		Test 2		Test 3		Test 4		Test 5	
	Manual	PPIR	Manual	PPIR	Manual	PPIR	Manual	PPIR	Manual	PPIR
all	886	0	62	0	62	0	62	0	62	0
cpath	886	917	346	349	2290	2293	369	372	354	357
fattr	886	917	346	349	2290	2293	369	372	354	357
rpath	886	917	346	349	2290	2293	369	372	354	357
stdio	886	921	346	353	2290	2297	369	376	354	361
wpath	886	917	346	349	2290	2293	369	372	354	357

Privilege	Test 6		Test 7		Test 8		Test 9		Test 10		Test 11	
	Manual	PPIR	Manual	PPIR	Manual	PPIR	Manual	PPIR	Manual	PPIR	Manual	PPIR
all	626	0	645	0	62	0	62	0	62	0	62	0
cpath	626	649	645	670	262	265	1165	1194	346	349	346	349
fattr	626	649	645	670	262	265	1165	1194	346	349	346	349
rpath	626	649	645	670	262	265	1165	1194	346	349	346	349
stdio	626	653	645	674	262	269	1165	1198	346	353	346	353
wpath	626	649	645	670	262	265	1165	1194	346	349	346	349

Table 6.6: Active Instructions per Privilege for cp

OpenBSD is an open source community driven operating system. The Pledge framework is developed and maintained by the community as part of the OpenBSD kernel. The utilities shipped as part of OpenBSD are also maintained by the same community and use calls to pledge to restrict privileges within programs. We compare the effectiveness of the manual instrumentation, performed by the maintainers, against the automated instrumentation performed by PPIR by examining the privileges available for each instruction executed. Since this information is only available at runtime, we use dynamic traces of the program generated by running the program against its test suite from BusyBox. Using the same test suite establishes trust that the privilege restrictions are not functionally incorrect.

Tables 6.3, 6.4, 6.5 and 6.6 show the effectiveness of PPIR compared to the manually instrumented version of cat, date, dd and cp respectively. Each table shows the number of instructions executed for each privilege available to the program for every test in the test-suite. The first column in the table are the privileges which were available to the program during its execution. The remainder of the columns show the number of instructions which were executed while the program held the privilege from the first column. Each of the columns correspond to a test in the test suite. Each test is executed by two versions: the OpenBSD version with privilege restrictions inserted manually by the authors, and the version produced by PPIR. For the latter, we remove the privilege restrictions made by the authors in the OpenBSD version before using PPIR to make the privilege restrictions. Note that the first row for every table represents information for the full set of privileges i.e. this row shows the number of instructions executed by the program prior to a single privilege restriction.

The manually instrumented version of cp allows the program to execute with the full set of privileges for three test cases (Tests 1, 6 and 7). PPIR is able to recognize the opportunity for a privilege restriction and perform restrictions for all three cases. The functional tests prove that this is a valid privilege restriction that reduces the privileges available per instruction.

We found that the version of date packaged with OpenBSD did not have any privilege restrictions while PPIR was able to infer the privileges required and perform restrictions.

The restrictions by PPIR were less effective on dd, and the instrumented version had significantly higher instruction counts for most privileges. We believe the poorer performance is due to the fact that our lowering strategy uses an approximation to find locations at whichc to lower. We explain this further in Section 6.3.2. The higher instruction counts arise from our methodology which we explain further in the next section. In all cases, the privileges we inferred were the same as those in the manually instrumented programs or better, as we explore in Section 6.3.2.

6.3 Methodology

Our tests compare the privilege restrictions made by the authors to the ones made by PPIR. Thus, for each benchmark program we use two versions of the program. The first is generated by compiling the original source code (with privilege restrictions from the authors) with *WLLVM*[32]. WLLVM produces a single module for the entire program which eases instrumentation and analysis. For the other version, i.e. the version in which PPIR performs the privilege restrictions, we remove the privilege restrictions made by the authors and compile it into a single module using WLLVM. Then for both versions, the produced modules are instrumented to produce a dynamic trace of instructions paired with the active privileges at that instruction.

To produce the dynamic trace, we split the program into regions that are segregated by calls to pledge. Each region is associated with the same set of privileges. After a call to pledge, the following region is limited to the set of privileges allowed by pledge. At runtime, the program produces a file with the instructions executed along with the privileges held while each instruction was executed.

Instrumenting the program to produce this data data proved to be challenging, as printing to stdout or stderr interferes with the test suite's oracle. Alternatively, opening and printing to a file necessitates the need for a privilege external to the semantics of the program. We work around this by opening a file in the global constructor of a program, and then printing to its descriptor for the remainder of the program. The semantics of pledge dictates that while the 'wpath' privilege is required to open a file, printing to it requires the 'stdio' privilege which is unconditionally required for all our benchmarks. The wpath

privilege is required at the global constructor but can be immediately removed at the start of the program.

Both versions of the program are instrumented to produce dynamic traces in the same fashion. The version of the program without privilege restrictions is then instrumented by PPIR for privilege restrictions. With the instrumentation finished, both versions are compiled into executables for OpenBSD and run against the test suite. While the control flow for both versions remained the same we found that the manually instrumented version had more optimizations and as a result fewer instructions. We mention possible improvements on the methodology to remove this discrepancy in Section 8.1.

6.3.1 Case Study 1: cat

cat is a command line utility that reads from either a file or from stdin and streams the contents to stdout. Listing 6.1 shows a basic block from cat after being analyzed and instrumented by PPIR. Line 4 is a call to fstat a system call that provides information about the status of file, successful calls to which return 0. The return value, stored in virtual register '%18', is compared to 0 on line 6. The boolean result of the comparison is used as an argument to the call to AdD_rpath. The AdD_X functions build a string specifying the privileges the program should retain at that point. Assuming the return value from fstat is 0, the rpath privilege will be added to the string. Additionally, the call to AdD_stdio on line 5 adds the stdio privilege. The argument to the call in this case is the boolean value true as the privilege is unconditionally required at that point.

```
1  ; <label>:17:      ; preds = %13
2    ...
3    call void @EnD_ReGiOn(...)
4    %18 = call i32 @fstat(...)
5    call void @AdD_stdio(i1 true)
6    %19 = icmp eq i32 %18, 0
7    call void @AdD_rpath(i1 %19)
8    call void @MaKe_pLeDgE()
9    %20 = icmp eq i32 %18, 0
10   call void @EnD_ReGiOn(...)
11   br i1 %20, label %23, label %21
12   ...
```

Listing 6.1: Hoisted privilege restriction in cat

Analyzing the instrumentation PPIR added to cat, we find that the conditions used in performing privilege restrictions were related to checking error conditions. The conditional restrictions made in cat as presented in Table 6.3 were made with the approximation for exit paths (Section 5.2.2) disabled. With the approximation enabled, the privilege requirements

become unconditionally true. This falls within the norm for command line utilities as they are built for a single responsibility with their privilege requirements being ubiquitous.

6.3.2 Case Study 2: dd

The dd utility performs reads and writes from specified input and output streams. Additionally, dd can be used for reading and writing from tape drives, which requires the privilege tape. PPIR is able to correctly identify a use of this privilege while performing privilege restrictions. The manually instrumented version uses an incorrect set of privileges and circumvents the operating system by using the error privilege. The error privilege allows a program to continue executing in the event that it uses system calls that it does not have access to. Table 6.5 shows that a large part of the program is allowed to execute with the error privilege enabled in the manual version of the program.

While PPIR was able to find more precise privileges for dd, it was not able to perform progressive privilege restrictions. We believe this is due to the fact the privilege using functions fstat and write are within a loop. Our analysis was not able to identify the point where the loop ended and perform conditional restrictions there since we use an approximation for the round-trip technique to select points of instrumentation. For the point to be identified, the round-trip check would have to be performed over the edge between the loop and its exit block. The analysis, in its present form, does not perform the round-trip check over edges and is unable to identify the potential point of instrumentation in dd.

6.4 Threats to Validity

Functional correctness of the instrumented programs, as demonstrated by the passing test suites run against the benchmarks, addresses the greatest concern for the validity of our analysis. As an initial implementation we believe bugs to be possible source of errors in our analysis despite significant efforts towards testing the program. The underlying privilege specifications which we provide to the static analysis may also have inaccuracies as they were reverse engineered without the input of the authors of the original system. Our static analysis supports the program structures which were encountered. In particular, PPIR supports branches, switch statements, functions, nested loops and recursive functions. For practical applications the static analysis must be extended to support threads, exceptions, setjmp() and longjmp(). We discuss this further in Section 7.1, Future Work.

Chapter 7

Related Work

This chapter details work in the area closest to ours. The related work can be categorized into work on precondition inference or program security and compartmentalization.

7.1 Security and Compartmentalization

Software compartmentalization reduces the available attack surface for a program and is an actively researched area and several frameworks are available for enabling compartmentalization[4] using privilege management. Our work is the first to automate compartmentalization using the Pledge framework.

PrivAnalyzer. Recently, Criswell et al. developed a toolchain called PrivAnalyzer[12] to perform privilege restrictions via static analysis and analyze their impact dynamically. *AutoPriv*[19] performs monotonic privilege reduction using the Capsicum capability framework for Linux. AutoPriv performs a liveness analysis[20] to find the points where privileges are no longer required and uses Capsicum primitives to enforce their restriction. The primary difference between AutoPriv and PPIR is that AutoPriv is flow-insensitive i.e. AutoPriv does not perform conditional privilege restrictions.

ChronoPriv[12] dynamically analyzes the program transformed by AutoPriv to find the number of instructions executed while holding privileges. We use a similar technique to assess PPIR against manually instrumented programs (Section 6.2). The report generated by AutoPriv is fed into *ROSA*[12], a bounded model checker which produces a risk assessment for the program.

API Permission Mapping. Aafer et al.[1] recently published work on reducing over privileging in mobile applications using permission specifications derived from analyzing the Android SDK. The specifications are maps of conditions on incoming arguments to operating system API calls which allow a finer granularity of control over required permissions. The arguments from an API call sites within an application are used to find the

conditions satisfied and the associated permissions without developer input. While their solution, dubbed *Arcade*, and ours share the same objective, i.e. reducing over-privileging in applications, our model can perform monotonic privilege restrictions. PPIR uses a novel dataflow analysis which differs from their graph abstraction technique to find preconditions.

Barring the incongruity of their operating domains, Arcade can be used complement to our work where the protection maps generated by Arcade can be used by PPIR to generate stronger specifications for privilege using functions.

Capweave. *Capweave*[17] accepts an input program and security policy specified in a specialized language. The policy associates a set of program points with the privileges that should be allowed at that point. Capweave instruments the input program with security primitives to achieve the security policy specified. Capweave uses the Capsicum capability system[39] to enforce restrictions on privileges. Unlike PPIR, the burden of analyzing and developing a security policy lies with the application developer. Furthermore, Capweave restructures the program, making it difficult to understand and therefore maintain for the programmer.

Program Partitioning. Rather than relying solely on the operating system to enforce privilege restrictions, programmers may architect their programs into components with fewer privileges available to each. *Privtrans*[9] partitions an input program into a monitor and slave. Similar to Privtrans, *ProgramCutter*[40] is another analysis driven tool for partitioning programs. While Privtrans requires the input program to be annotated to indicate the privilege using functions, ProgramCutter requires no external input aside from the original program.

Wedge[7] is another tool which can be used by programmers to compartmentalize and execute privileged code on secure threads using the SELinux[37] security module. Wedge also provides an associated tool, *Crowbar*, to aid programmers in compartmentalizing their programs. PPIR is set apart from these, and other similar tools[9][22], since it does not require the program to be restructured to enable compartmentalization. It also does not require additional effort from the programmer.

Secure Compilation. It is desirable to have programs compiled with safety properties. A safety property, supplied at compile time, guarantees that the program will not be subject to the attack or threat the property protects against. For example, a program written in a memory safe language (such as Java or C#) is guaranteed to be free of memory errors as defined by the source language. A safety property can provide guarantees against entire classes of attacks. Secure compilation guarantees the preservation of source level safety properties when during compilation. It does not attempt to imbue the program with

additional safety guarantees or mitigations. PPIR, on the other hand, attempts to provide an additional safety feature(i.e. privilege restrictions) to a program written without it.

Fully Abstract Compilation(FAC) is a technique[2][30][15] which preserves observational equivalence between the source and target language during compilation, preserving safety properties. Robustly Safe Compilation[31], similar to FAC, discards guarantees for properties which are not strictly required for safety.

Abate et al.,[3] propose a formal criterion and a supporting compiler toolchain for ensuring additional compartments in compartmentalized programs are not compromised when one is compromised at runtime. While the objectives of PPIR are different from the secure compilation, securely compiled programs can benefit from the additional safety provided by PPIR.

7.2 Precondition Inference

While precondition inference techniques have been applied to several domains, our work is a novel approach which applies precondition inference for program security.

Null Derefence Verification. Madhavan et al.[25] describe an approach to synthesizing over approximated weakest preconditions to verify dereferences in Java. The approach builds a logical formula from a single dereference to the program entry point. The analysis is built on Kildall's[20] dataflow analysis framework, using fixed point iteration to compute weakest preconditions over loops and recursive functions. In contrast, our approach computes the Strongest Necessary Preconditions to preserve the soundness of our analysis. The computed preconditions are then used for program hardening which requires program instrumentation, adding a dynamic component to our work.

Necessary Precondition Inference. Cousot et al.[10] propose a tool for precondition inference to generate contracts based on their previous work[11]. The generated contracts are used to guarantee assertions within the programs to prevent faulty executions. Our work builds on theirs and introduces the idea of using a single step forward postcondition analysis to find locations to insert conditional privilege restrictions.

The research effort in this area is weighted towards security while maintainability is of lower priority. PPIR provides a middle ground, sacrificing security guarantees which can only be made by refactoring programs in favor of long-term maintenance. Additionally, PPIR is unique in leveraging the Pledge framework for compartmentalization preventing direct comparisons with other tools.

Chapter 8

Conclusions and Future Work

This thesis presents a first attempt at a novel static analysis technique built to automatically infer the privileges required by a program and hoist the conditions required for it to perform progressive conditional privilege restrictions. We are able to show the benefits of precise automated privilege inference comparable with manual privilege restrictions. Our results show that the analysis is able to hoist constraints extracted from the program to restrict privileges conditionally. At the heart of our work lies a novel postcondition to precondition round-trip check that highlights the points in the program where inserting guarded privilege reductions may be beneficial.

8.1 Future Work

As an initial attempt at such an analysis we believe there is plenty of room for future work in terms of improving the analysis, methods of approximation and new avenues of research which can be performed using such an analysis technique.

Scaling to Larger Programs Despite the simplicity of the technique used, we were able to scale the analysis using our approximations for an OpenBSD network utility, the *stateless address autoconfiguration daemon* (slaacd). The network relies on the libevent library, which as its name suggests is a library that provides a framework for implementing event driven programs. PPIR was able to extract the required through the event queue driver. However, the privileges required by the program were outside of the privileges which we support (as of OpenBSD 6.5, slaacd uses experimental undocumented privileges) and testing the validity of our results proved to be difficult. We believe that the fact the analysis was able produce privilege requirements through an event driven framework shows promise as a potential class of programs which can be analyzed with our technique.

Program Structures Threads and forks are ubiquitous in modern programming and support for these programming structures is necessary for performing analyses on servers and applications. PPIR does not currently support threads or forks and can be extended to support them. Splitting an application into sub-processes is an effective technique for compartmentalization[16][7]. Using the Pledge framework, developers can limit the privileges of sub-processes using the second argument (`const char* execpromises`) to a `pledge` call. When the second argument is left unspecified, sub-processes are allowed to execute with access to all privileges and should call pledge on their own. A program compartmentalized into independent processes would also compose well with our analysis technique. Conservatively, we can model inter-processes communications as `havoc` statements similar to external inputs. This could be improved by generating protection maps for inter-process communications similar to Arcade[1].

Threads, while functionally similar to processes, pose a different issue since they exist within the same process and share privileges. Necessary conditions for system calls within parallel threads would have to be checked prior to the parallel code along with any privilege restrictions. Shared or aliasing memory within separate threads would also affect the conditions and must be dealt with. Another approach would be to use a default-deny model as demonstrated in Wedge[7].

Analysis Technique The analysis as implemented uses a naive call strings based approach to perform the dataflow analysis. Existing function summary based approaches would make the analysis more efficient possibly allowing it to scale to larger programs. We tested our current implementation against a `pax`, a utility similar to `cp` which can produce archives. Unfortunately, the analysis ran out of memory and was unable to complete.

SMT Based Simplifications The simplifications techniques which we use for minimizing the constraint set do not incorporate any of the knowledge available in the underlying constraints aside from the constant simplification technique. Satisfiability Modulo Theoretic(SMT) solvers such as Z3[13] can use the constraints generated during the precondition inference step through a first-order language to perform reductions which would be far more powerful for simplifications.

Approximation Methods Null checks dominate the constraints when performing precondition inference and have been shown to have little effect on the precision of the results [25]. We believe that there are more precise approximations to be made in place of the naive methods currently in play.

Context Sensitive Lowering While the analsis is able to produce context sensitive constraints, the lowering technique simply merges them across all contexts via the meet operator. The analysis loses precision at this point and better precision could be achieved by making the instrumentation context aware.

Bibliography

[1] Yousra Aafer, Guanhong Tao, Jianjun Huang, Xiangyu Zhang, and Ninghui Li. Precise android api protection mapping derivation and reasoning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1151–1164. ACM, 2018.

[2] Martín Abadi. Protection in programming-language translations. In *Secure Internet programming*, pages 19–34. Springer, 1999.

[3] Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Catalin Hritcu, Théo Laurent, Benjamin C Pierce, Marco Stronati, and Andrew Tolmach. When good components go bad: Formally secure compilation despite dynamic compromise. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1351–1368. ACM, 2018.

[4] Jonathan Anderson. A comparison of unix sandboxing techniques. *FreeBSD Journal*, 2017.

[5] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.

[6] David Barrera, H Güneş Kayacik, Paul C Van Oorschot, and Anil Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 73–84. ACM, 2010.

[7] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. In *na*. USENIX Association, 2008.

[8] David Brumley and James Newsome. Alias analysis for assembly. Technical report, Technical Report CMU-CS-06-180, Carnegie Mellon University School of âĂę, 2006.

[9] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium*, pages 57–72, 2004.

[10] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. Automatic inference of necessary preconditions. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 128–148. Springer, 2013.

[11] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. Precondition inference from intermittent assertions and application to contracts on collections. In *International*

Workshop on Verification, Model Checking, and Abstract Interpretation, pages 150–168. Springer, 2011.

[12] John Criswell, Jie Zhou, Spyridoula Gravani, and Xiaoyu Hu. Privanalyzer: Measuring the efficacy of linux privilege use. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 593–604. IEEE, 2019.

[13] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[14] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.

[15] Daniele Gorla and Uwe Nestmann. Full abstraction for expressiveness: History, myths and facts. *Mathematical Structures in Computer Science*, 26(4):639–654, 2016.

[16] Munawar Hafiz, Ralph Johnson, and Raja Afandi. The security architecture of qmail. In *Proceedings of the 11th Conference on Patterns Language of Programming (PLoPâĂŹ04)*. Citeseer, 2004.

[17] William R Harris, Somesh Jha, Thomas Reps, Jonathan Anderson, and Robert NM Watson. Declarative, temporal, and practical programming with capabilities. In *2013 IEEE Symposium on Security and Privacy*, pages 18–32. IEEE, 2013.

[18] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[19] Xiaoyu Hu, Jie Zhou, Spyridoula Gravani, and John Criswell. Transforming code to drop dead privileges. In *2018 IEEE Cybersecurity Development (SecDev)*, pages 45–52. IEEE, 2018.

[20] John B Kam and Jeffrey D Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977.

[21] Paul A Karger. Limiting the damage potential of discretionary trojan horses. In *1987 IEEE Symposium on Security and Privacy*, pages 32–32. IEEE, 1987.

[22] Douglas Kilpatrick. Privman: A library for partitioning applications. In *USENIX Annual Technical Conference, FREENIX Track*, pages 273–284, 2003.

[23] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323–337, 1992.

[24] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[25] Ravichandhran Madhavan and Raghavan Komondoor. Null dereference verification via over-approximated weakest pre-conditions analysis. In *ACM Sigplan Notices*, volume 46, pages 1033–1052. ACM, 2011.

[26] Gary McGraw. Software security. *IEEE Security & Privacy*, 2(2):80–83, 2004.

[27] Anders Møller and Michael I Schwartzbach. Static program analysis. *Notes. Feb*, 2012.

[28] Diego Novillo et al. Memory ssa-a unified approach for sparsely representing memory operations. In *Proc of the GCC DevelopersâĂŹ Summit*. Citeseer, 2007.

[29] OpenBSD. Openbsd man page for pledges. `https://man.openbsd.org/pledge.2`.

[30] Joachim Parrow. General conditions for full abstraction. *Mathematical Structures in Computer Science*, 26(4):655–657, 2016.

[31] Marco Patrignani and Deepak Garg. Robustly safe compilation. In *European Symposium on Programming*, pages 469–498. Springer, 2019.

[32] T. Ravitch. Whole program llvm. `https://github.com/travitch/whole-program-llvm`.

[33] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sep. 1975.

[34] Fred B Schneider. Least privilege and more [computer security]. *IEEE Security & Privacy*, 1(5):55–59, 2003.

[35] M Sharir and A Pnueli. *Two approaches to interprocedural data flow analysis*. New York Univ. Comput. Sci. Dept., New York, NY, 1978.

[36] Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Citeseer, 1991.

[37] Stephen Smalley, Chris Vance, and Wayne Salamon. Implementing selinux as a linux security module. *NAI Labs Report*, 1(43):139, 2001.

[38] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266. ACM, 2016.

[39] Robert NM Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for unix. In *USENIX Security Symposium*, volume 46, page 2, 2010.

[40] Yongzheng Wu, Jun Sun, Yang Liu, and Jin Song Dong. Automatically partition software into least privilege components using dynamic data dependency analysis. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 323–333. IEEE, 2013.