UNIVERSITY OF CALIFORNIA,
IRVINE

Security Applications of Static Program Analysis

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Brian Jon Belleville

Dissertation Committee:
Professor Michael Franz, Chair
Professor Ian G. Harris
Professor Alexandru Nicolau

2018

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ACKNOWLEDGMENTS

Above all else, I would like to thank my advisor Professor Michael Franz. My time at UC Irvine has been a period of great professional and personal development, and I am extremely grateful to have been given this opportunity. None of this would have been possible without your support and guidance, and the excellent research group you have created.

I would also like to thank Professor Ian Harris and Professor Alexandru Nicolau for serving on my doctoral committee.

I have also been fortunate to have been supervised by excellent postdocs in our research group. Thank you to Dr. Stefan Brunthaler, Dr. Per Larsen, Dr. Stijn Volckaert, and Dr. Yeoul Na for all of your help. Your feedback and contributions on the papers, presentations, and research projects I have worked on has been incredibly valuable, and I have learned a great deal from each of you.

Another important part of my experience at UC Irvine has been all of the other graduate students in our lab, you have been a large part of what has made this time both productive and fun. Thank you all for your friendship and the many interesting discussions we have had, especially: Julian Lettner, Taemin Park, Paul Kirth, Prabhu Rajasekaran, Anil Altinay, Joe Nash, Alexios Voulimeneas, Mohaned Qunaibit, Dokyung Song, Codruţ Stancu, Andrei Homescu, and Stephen Crane.

I would also like to thank all of my collaborators outside of UC Irvine. Thank you to Professor Yunheung Paek and his students Hyungon Moon, Jangseop Shin, Dongil Hwang, and Seonhwa Jung for the collaboration on Hardware Assisted Randomization of Data, and thank you to Wenbo Shen and Ahmed Azab for the collaboration on the Kernel Address Leak Detector.

This research is based on work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-15-C-0124 and FA8750-15-C-0085, the National Science Foundation under awards CNS-1619211 and CNS-1513837, the United States Office of Naval Research (ONR) under contract N00014-17-1-2782, as well as gifts from Mozilla, Oracle, and Qualcomm.

Portions of this dissertation have been accepted, and are soon to be published by Springer International Publishing, who owns the respective copyrights.

CURRICULUM VITAE

Brian Jon Belleville

EDUCATION

Doctor of Philosophy in Computer Science **2018**
University of California, Irvine *Irvine, California*

Master of Science in Computer Science **2015**
University of California, Irvine *Irvine, California*

Bachelor of Science in Physics **2011**
University of California, Los Angeles *Los Angeles, California*

RESEARCH EXPERIENCE

Graduate Research Assistant **2013–2018**
University of California, Irvine *Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant **2014–2015**
University of California, Irvine *Irvine, California*

REFEREED CONFERENCE PUBLICATIONS

Hardware Assisted Randomization of Data **2018**
International Symposium on Research in Attacks, Intrusions and Defenses

ABSTRACT OF THE DISSERTATION

Security Applications of Static Program Analysis

By

Brian Jon Belleville

Doctor of Philosophy in Computer Science

University of California, Irvine, 2018

Professor Michael Franz, Chair

Static program analysis computes information about a program without executing the program. This can be used to improve software security by determining a security policy based on the program's semantics, which is then used to implement a run-time protection, or by detecting bugs in the program, which can then be fixed before they are subject to an attack. We present applications of static program analysis to address software exploits that utilize memory corruption.

Memory corruption exploits are one of the most severe forms of program attacks, and occur when an attacker performs invalid memory accesses to hijack a program. This can involve overwriting data to force the program to perform malicious actions, as well as reading sensitive data to leak secrets.

One approach that has been proposed to stop memory corruption attacks is data space randomization (DSR). DSR utilizes static analysis to classify program variables into a set of equivalence classes, and then encrypts variables with a randomly chosen key for each equivalence class. This thwarts memory corruption attacks that introduce illegitimate data flows. However, existing implementations of DSR trade analysis precision for better run-time performance, which leaves attackers sufficient leeway to mount attacks. In this dissertation we present context-sensitive data space randomization, a more precise version of DSR that is

able to distinguish a larger number of equivalence classes by using a context-sensitive points-to analysis to construct equivalence classes. We then adapt this analysis and protection to HARD, which shows that context-sensitive DSR can target specialized hardware to provide precise protection with good run-time performance.

We also explored using static analysis to find security critical bugs. Specifically, we developed `KALD`, a static analysis tool which uses points-to analysis to detect direct address disclosures that can lead to kernel ASLR bypasses. We show that `KALD` successfully detects several previously unknown direct disclosure vulnerabilities in the Linux kernel.

Chapter 1

Introduction

Static program analyses are algorithms that determine properties about a program without actually running the program. One fundamental analysis is points-to analysis, which provides answers to queries about what objects a pointer can refer to. There has been extensive research on this topic to address issues such as precision and scalability [5, 96, 51, 70, 9, 109, 103, 112, 98, 34]. The underlying analysis results can be used by different clients for a variety of applications, including security applications. Points-to analysis has previously been used to compute a security policy that becomes a component of run-time defenses [11, 15, 17, 3] as well as to detect bugs in programs before they can be exploited [8, 45, 76]. In this dissertation we explore applications of static points-to analysis that address security issues caused by memory corruption.

Memory corruption occurs when a programming error allows an attacker to modify memory in a way that is not intended by the programmer. This can lead to powerful and expressive attacks where, in the worst case, an attacker is able to execute arbitrary code of their choosing. Programs written in low-level systems programming languages such as C and C++ are vulnerable to these types of errors. There are alternative programming languages that

enforce memory safety, and if they are used, memory corruption can be eliminated. However unsafe languages are still widely used for their performance and fine grained control. There are many important applications written in low-level programming languages including web servers, internet browsers, operating system kernels, and word processors, and it would require substantial development effort to rewrite these applications in a language that enforces memory safety. Since memory corruption exploits are severe, there is a large volume of research in methods to prevent these attacks and develop new exploitation techniques that avoid the prevention mechanisms [99].

Enforcing run-time memory safety of programs written in unsafe languages will prevent memory corruption, however this often comes with unacceptable performance degradation. An alternative to memory safety enforcement is exploit mitigation. Exploit mitigation techniques minimize the impact of memory corruption vulnerabilities, but they do not remove the underlying vulnerabilities. Exploit mitigations are able to raise the bar for attackers, and many are widely adopted.

Most exploit mitigations focus on mitigating attacks that perform control data overwrites. Control data is any data value that will be used as the value of the instruction pointer at some point. Examples of control data are return addresses and function pointers. Control flow integrity (CFI) [2] mitigates control data overwrites by restricting the targets of indirect control flow to the set of valid targets intended by the programmer. This is effective against an attack that corrupts control data and changes the control flow, but it does not prevent an attacker from performing a memory overwrite, or protect any other data. An attacker is still able to overwrite non-control data to exploit a program [20, 56, 57].

Data space randomization (DSR) has been proposed as a mitigation against non-control data attacks [11, 15]. DSR randomizes the representation of data stored in memory in order to make it difficult for an attacker to control the result of an overwrite. This is applied to all data in a program, and is able to mitigate attacks on both control and non-control data.

DSR relies on the result of static points-to analysis to determine what data transformations are safe to make. This analysis is conservative, and imprecision will lead to reduced security. Additionally, prior DSR techniques made several design decisions that reduce the security of the protection. In this dissertation we present context-sensitive data space randomization, an extension to DSR that uses a more precise, context-sensitive analysis algorithm to provide greater levels of security.

While context-sensitive DSR is able to provide more precise protection, it comes at the cost of higher run time overhead. Also, there are still limitations common to all DSR techniques implemented purely in software. To improve run time overheads and address other limitations of DSR, we present hardware assisted randomization of data (HARD), a hardware assisted implementation of context-sensitive DSR that reduces run time overheads and utilizes hardware support to further harden the DSR implementation.

One exploit mitigation that has found widespread adoption is address space layout randomization (ASLR) [85]. ASLR randomizes the base address of different sections of a program, and it has been adopted by all major operating systems for both user-space applications and the operating system kernel. The primary weakness of ASLR is that if the randomization offset is discovered, an attacker can then bypass the protection. To maintain protection it is critical that this offset is protected, and this is especially true for operating system kernels since the randomization offset is not changed until the system is restarted. To address ASLR offset leaks we present the Kernel Address Leak Detector (`KALD`), a static analysis tool for finding direct disclosure vulnerabilities within the context of operating system kernels. Our tool is able to analyze the source code of operating system kernels and detect locations where a developer may inadvertently reveal an address that will leak the randomization offset.

Chapter 2

Background

In this dissertation we investigate ways to prevent exploits that utilize memory corruption. Memory corruption occurs in programs written with unsafe, low-level languages like C and C++. These languages do not verify that a memory access is valid at run-time. Instead, performing an invalid memory access is considered to be undefined behavior, and the responsibility is on programmers to ensure their programs do not exhibit such behavior. However for complex applications this task is difficult, and memory corruption vulnerabilities are some of the most commonly reported software vulnerabilities [22].

Memory corruption can occur in several forms, the most straightforward is a buffer overflow. A buffer overflow occurs when the program erroneously writes data to a memory buffer that exceeds the size of the buffer. This will overwrite the adjacent memory following the buffer and can be used to corrupt variables stored adjacent to the buffer. A more powerful form of memory corruption is when an attacker can directly control the memory address that is accessed. This allows the attacker to perform arbitrary memory writes, enabling them to overwrite any program variable, instead of being limited to those adjacent to a vulnerable buffer. Both buffer overflow and arbitrary write are spatial errors, the attacker is writing to

locations that are not allowed. There are also temporal memory errors, where an attacker is able to access memory outside of its lifetime. One common temporal memory error is use-after-free. In a use-after-free, due to a programming error, a pointer to an object still remains after the object has been freed. An attacker can use this so-called dangling pointer to alter the memory after it has been returned to the memory allocator. The memory allocator may reuse the memory of the freed object and place a different variable at the same location. When an attacker writes to memory through the dangling pointer, they will alter the value of this newly allocated variable.

2.1 Attacks and Defenses

Memory corruption is the building block that an attacker uses to craft an exploit. Simply causing a memory error to occur is usually not enough to cause the program to perform the action the attacker desires. Attackers have developed a number of sophisticated methods to craft attacks using memory corruption. This has sparked a sort of cat and mouse game between attackers and defenders, when attackers discover an exploit technique, new defenses are developed to prevent that exploit, and in response, attackers find new ways to exploit programs [99].

One of the most direct exploitation techniques is code injection. In a code injection attack, the attacker fills a buffer with machine code instructions. The attacker then causes the program to execute this code, for example by overwriting a return address with the address of the buffer [4]. This is only possible on systems where data can be executed. The attacker writes the exploit code to a data buffer, which is then interpreted as instructions when execution is directed to that location. Modern operating systems use hardware-enforced memory permissions to separate code and data using the policy that a memory page cannot be both executable and writable [6]. This policy is called W$\oplus$X or data execution prevention

(DEP).

DEP stops code injection, but attackers can still craft exploits using the code already present in a program. This is known as a code reuse attack. A notable type of code reuse attack is return-oriented programming (ROP) [91]. In a ROP attack, the attacker finds short sequences of instructions, called gadgets, that end with a return instruction. The attacker searches the code of the program to find gadgets, and then constructs a sequence of gadgets that performs their desired attack. The attacker writes this sequence of gadget addresses to memory they control, then changes the stack pointer to force the program to interpret these addresses as a sequence of return addresses. This allows for extremely expressive attacks. Many programs contain a Turing complete set of gadgets [91], meaning the attacker can construct gadget chains to perform arbitrary computations. There are also many other forms of code reuse such as return into libc [83], which reuses whole functions in the standard library, jump-oriented programming [12], which uses indirect branch instructions instead of return instructions, and counterfeit object-oriented programming [88], which abuses C++dynamic dispatch.

One widely adopted mitigation against code reuse attacks is address space layout randomization (ASLR) [85]. ASLR introduces randomness to the memory layout of a program by randomly selecting the base address of each memory segment used by the program. ASLR has been adopted by all major operating systems for both user programs and the operating system kernel. It is effective at stopping code reuse attacks because code reuse attacks require the attacker to know the exact code layout of the program. For example, a ROP attack requires the attacker to know the addresses of all the gadgets they wish to execute. With sufficient entropy, the likelihood an attacker will randomly guess the correct addresses is very low. Therefore, in order to mount an attack against a system protected by ASLR, the attacker would first need to discover the memory layout of the code. However since ASLR applies only a single offset to an entire memory section, if the attacker can discover

the address of one item within that section, they can determine the offset and know the layout of the entire section. In practice, attackers are often able to discover the ASLR offset and bypass the protection [90].

Although code reuse uses exiting code in the application, the execution paths are very different from valid runs of the program. Code reuse attacks lead to erroneous control flow that was not intended by the programmer. Since the control flow is dramatically changed by the attack, enforcing control flow integrity (CFI) [1] is an an effective mitigation against code reuse attacks. CFI restricts the targets of indirect control flow instructions to the set of targets allowed in valid executions of the program. This is enforced by inserting checks before each indirect control transfer instruction. When the attacker attempts to divert control flow to malicious paths, the checks will fail, and the attack will be stopped. CFI has now been adopted by mainstream compilers, both gcc and LLVM now implement CFI enforcement [100].

CFI and ASLR primarily protect against attacks that corrupt control data. Control data is any data that becomes the value of the instruction pointer at some point in the program. Examples of control data are return addresses or function pointers. Code reuse attacks must overwrite control data to divert the control flow and perform the attack. However, there exist another class of attacks, called non-control data attacks, which do not violate control flow integrity or depend on code layout. Non-control data attacks are any attacks that do not overwrite control data, and therefore do not introduce erroneous control flow. Instead non-control data attacks overwrite other security-critical data in order to exploit the system. An example is corrupting the variable that specifies the CGI-BIN directory of a web server to allow the attacker to execute arbitrary commands. Non-control data attacks can be just as severe as control-data attacks [20], and can be equally expressive [57], allowing the attacker to perform Turing complete computations. Non-control data attacks may still depend on the memory layout, so ASLR does mitigate these attacks to some extent. However, ASLR

still suffers from the limitation that it can be bypassed by leaking a single pointer.

Data space randomization (DSR) [11, 15] is a promising technique that has been proposed to mitigate non-control data attacks. DSR randomizes both control and non-control data stored in memory to provide probabilistic protection against exploits. The randomization is accomplished by analyzing a program's source code and classifying all variables into equivalence classes. Each equivalence class is then assigned a different encryption key. All memory accesses are instrumented to encrypt data before stores and decrypt data after loads by XORing the data with the correct key. The equivalence classes are selected such that for each load or store instruction, all valid targets of that instruction will be placed in the same equivalence class. Since all valid targets will use the same key, the program will function correctly during normal executions. However, if an attacker overwrites an object that is not a valid target of an instruction, the encryption key for the invalid target will be different. Since the attacker does not know the keys, they will not be able to control the result of the overwrite. With sufficient key entropy this provides effective probabilistic protection against memory corruption attacks.

To classify variables into equivalence classes, DSR must know which objects are valid targets of all load and store instructions. Pointer aliasing in C makes determining the set of allowed targets a difficult problem so DSR uses points-to analysis to compute the set of valid targets of load and store instructions and construct equivalence classes.

2.2 Points-to Analysis

Points-to analysis is a static program analysis that computes the set of objects a pointer can refer to [93]. There is a large amount of research on points-to analysis, and a complete background on this topic is outside the scope of this dissertation. We provide background

```
int a, b;
int *x, *y;

x = &a;
y = &b;
x = y;
```

Listing 2.1: A simple sequence of C assignment statements.

on the relevant aspects of pointer analysis used by our work.

Points-to analysis can compute the set of objects a pointer can refer to, this can be used for many applications, including constructing the equivalence classes used by DSR. We would like to know the exact set of objects a pointer can refer to at run-time, however pointer analysis is a fundamentally difficult problem, and computing a fully precise result is undecidable [66]. Consequently, all algorithms produce an approximation of a fully precise result. For many applications, including DSR, we must know all valid targets, but it is acceptable to overestimate the set of targets. Algorithms that overestimate the set of targets are known as may-analyses. For DSR, overestimating the set of valid targets will cause some unrelated objects to be encrypted with the same key, but all valid targets are guaranteed to be in the correct class, so the program will still function correctly with DSR applied. While many algorithms have been developed that produce an over-approximation [5, 96, 34, 70], some algorithms are more precise and able to more closely approximate the fully precise result.

In points-to analysis algorithms, there is generally a trade off between analysis precision and analysis cost. There are several different characteristics of an analysis algorithm that determines where it falls on this continuum. One defining aspect of an analysis is how it processes assignment statements. The main distinction is between algorithms where the points-to set of the source becomes a subset of the points-to set of the destination, and algorithms where the points-to sets of the source and destination are unified. Andersen's

```
void foo() {
  int a = 0;
  int b = 1;
  bar(&a);
  bar(&b);
}

void bar(int *x) {
  *x++;
}
```

Listing 2.2: An example of the difference between context-sensitive and context-insensitive analysis.

analysis [5] is the prototypical subset based analysis while Steensgaard's analysis [96] is the prototypical unification based analysis. Listing 2.1 shows a sequence of assignment statements that lead to different results for different algorithms. A subset based analysis will compute different points-to sets for `x` and `y`, where `x` may point to `a` or `b` and `y` may point to only `b`. A unification based analysis will unify the points-to sets of `x` and `y` when the final assignment is analyzed, so the result would be that both `x` and `y` may point to `a` or `b`. Generally, subset based analyses provide more precise results than unification based analyses, however unification is generally more efficient. Whether the increased precision is useful depends on the client of the analysis. For the specific application of data space randomization, Bhatkar and Sekar show that a unification based analysis and a subset based analysis will produce the same set of equivalence classes [11].

Another variability axis that can increase the precision of an algorithm is context sensitivity. Context sensitivity refers the ability to increase precision by modeling different program states, and computing different results for different states. A context insensitive algorithm, in contrast, computes a single result that is valid for any possible program state. Context sensitivity generally models the calling context in a program, and computes different results for different call paths. An example of a program that benefits from a context-sensitive analysis is shown in Listing 2.2 . The function `bar` is called with the address of both `a`

and `b` as arguments, but otherwise `a` and `b` are unrelated. A context insensitive analysis will determine that the points-to set of `bar`'s formal argument `x` contains both `a` and `b`. A context-sensitive algorithm is able to distinguish between different calling contexts, and will determine that from the first call site the points-to set of `x` contains `a` and from the second call site it contains `b`. While a context-sensitive analysis provides a more precise result, prior DSR approaches are unable to utilize the increased precision because they have no way to dynamically choose the encryption key based on the run-time calling context.

Chapter 3

Context-Sensitive Data Space Randomization

3.1 Motivation

DSR encrypts variables that are stored in the program's memory, and it uses different keys to encrypt unrelated variables. This encryption makes the results of load and store operations that violate the program's intended data flow unpredictable, and thus hinders reliable construction of attacks, including non-control data attacks. However, prior work on DSR makes several trade-offs that favor run-time performance over security. First, existing versions of DSR do not encrypt variables that cannot be used as the base of an overflow attack. This leaves programs unprotected against temporal memory exploits such as use-after-free or uninitialized reads. Second, prior versions often use weak encryption keys to avoid the cost of handling unaligned memory accesses. Lastly, existing implementations rely on imprecise program analyses, which lead them to incorrectly classify many variables as related. As a result, these unrelated variables are encrypted with the same keys. Many unintended data

flows are therefore still possible, which gives attackers some leeway to construct exploits.

This motivated our work on context-sensitive data space randomization. Context-sensitive DSR offers greater security than prior approaches by distinguishing more unrelated variables. To do this, context-sensitive DSR uses a context-sensitive points-to analysis and generates encryption operations that use calling context specific keys. Context-sensitive DSR also encrypts *all* of the program data, and consistently uses strong 64-bit encryption keys. Thus, unlike existing schemes, context-sensitive DSR does not compromise its security guarantees for better run-time performance.

3.2 Background

Our goal is to thwart attacks that violate the intended data flow of a program. Listing 3.1 illustrates two such violations: a *use-after-free* and an *uninitialized read*. Both types of unintended data flows are examples of temporal memory errors and are highly relevant in practice. Use-after-free is commonly exploited to attack high-profile targets such as web browsers and operating system kernels [86], and the well-known Heartbleed bug was, at its core, an uninitialized read vulnerability [23].

At lines `(a-1)` and `(a-2)` in the example, the program allocates and initializes a list, `X`, as depicted in Figure 3.1-(a). At line `(b-1)`, the program frees the second element of list `X`, so the `Next` member of the first element becomes a dangling pointer. The program then allocates a new list, `Y`, at line `(b-2)`. The program now reads the contents of list `Y` without initialization at line `(b-3)`. Due to the deterministic nature of common memory allocators such as *dlmalloc* [71], the two lists will likely be laid out in the memory as shown in Figure 3.1-(b). Thus, the data read at line `(b-3)` will likely include the recently freed element of list `X`.

```
struct list {
  struct list *Next;
  int Data;
};

list *makeList(int Num) {
  list *New = new list;
  New->Next = Num ? makeList(Num-1) : 0;
  return New;
}

void fillList(list* L, int base) {
  if(L->Next)
    fillList(L->Next, base+1);
  L->Data = base;
}

void dumpList(list* L) {
  for (list* T = L; T->Next; T = T->Next)
    printf("%d\n", T->Data);
}

int main(int argc, char** argv) {
  list *X = makeList(4);     // (a-1)
  fillList(X,10);            // (a-2)
  delete X->Next;            // (b-1)
  list *Y = makeList(3);     // (b-2)
  dumpList(Y);               // (b-3)
  fillList(Y,20);            // (c-1)
  dumpList(X);               // (c-2)
  return 0;
}
```

Listing 3.1: A synthesized program illustrating use-after-free and uninitialized read vulnerabilities.

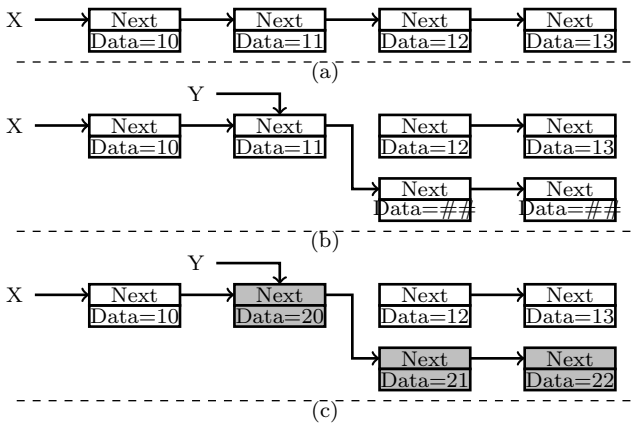

Figure 3.1: The diagram shows the lists generated in Listing 3.1. (a) shows list `X` after initialization at line `(a-2)`. (b) shows the most likely layouts of lists `X` and `Y` at line `(b-3)`. (c) shows the most likely layouts of the lists at line `(c-2)`.

The rest of the example demonstrates the use-after-free vulnerability. The program attempts to print the contents of list `X`, whose second element was freed at line `(b-1)`. A deterministic memory allocator may allocate the list `X` as shown in Figure 3.1-(c), and the dumped list includes elements of list `Y`.

3.2.1 Mitigation with DSR

DSR mitigates such unintended data flows by randomizing the representation of program data in memory. DSR relies on alias analysis to compute the points-to relations between pointers and the storage locations they can reference. Two pointers are considered aliases if they can reference the same storage location. Similarly, a pointer p may alias named object o, if p can point to o. Based on the alias analysis, DSR partitions storage locations into *equivalence classes* so that all storage locations belong to an equivalence class. Any two storage locations that may alias each other belong to the same equivalence class.

DSR encrypts storage locations belonging to different equivalence classes with distinct encryption keys. Locations belonging to the same equivalence class, however, must be encrypted with the same key. In the previous example, an ideal implementation of DSR would

see that lists `X` and `Y` are disjoint, and would encrypt them with different keys. An attacker that does not know the keys cannot extract the true contents of the illegally read list elements.

Unfortunately, existing implementations of DSR cannot prevent the exploits in this example [11, 15]. They *do* consider lists `X` and `Y` related because of the imprecise *(context-insensitive)* alias analysis which does not consider the functions' calling contexts. In the example, both `X` (at line `(a-2)`) and `Y` (at line `(c-1)`) are passed as an argument to `fillList`, and the context-insensitive alias analysis will report that the formal argument `L` of `fillList` may alias both `X` and `Y`. Variables `X` and `Y` will therefore be assigned to the same equivalence class.

Context-sensitive data space randomization avoids this loss of precision by using a *context-sensitive* alias analysis. If we analyze our example program with a context-sensitive alias analysis, we obtain two sets of aliasing relations: one for the calling context at line `(a-2)` where `fillList`'s formal argument `L` aliases `X`, and one for the calling context at line `(c-1)` where `L` aliases `Y`. By taking the calling context into account, we avoid having to treat `X` and `Y` as aliases and can therefore place them in different equivalence classes.

Leveraging the greater precision of context-sensitive alias analyses is challenging since the DSR instrumentation code must take the calling context into account to determine which encryption key should be used. We discuss this challenge at length in section 3.3, and present a novel DSR scheme that supports different contexts via dynamic key binding.

The vulnerabilities presented in Listing 3.1 also demonstrate the need to encrypt all equivalence classes, not just those that can be the base of an overflow. Both use-after-free and uninitialized read are temporal memory errors. A DSR implementation that only encrypts classes that may have spatial memory violations would not encrypt the lists `X` and `Y`. Even a context-sensitive DSR implementation that can place `X` and `Y` in different equivalence classes

will not mitigate the memory errors in this program if both lists are not encrypted.

3.3 Design

We begin this section by providing a conceptual overview of our design, and then discuss several key components in detail. The primary goal of context-sensitive DSR is to increase the precision of DSR by using a context-sensitive pointer analysis to construct equivalence classes. At a high level our system functions similarly to other DSR systems. We first perform a pointer analysis, then use the results of the pointer analysis to classify all objects in the program into different equivalence classes and assign unique keys to each class. Finally we transform the statements of the program such that a value will be encrypted before it is stored to memory, and decrypted after it is loaded from memory.

Context-sensitive DSR transforms input programs at the compiler intermediate representation (IR) level. The first step is a context-sensitive pointer analysis that categorizes the program's memory locations into equivalence classes based on the points-to sets computed by this analysis. We then assign two types of keys to the memory access instructions in the program, according to the equivalence classes they access. We assign a *static key* to instructions that always access the same equivalence class, regardless of its calling context, and a *dynamic key* to the others, which may access multiple equivalent classes depending on the calling context. The static keys are directly embedded into the program as constants so that each instruction can fetch its key, while dynamic keys are passed to a callee through the *context frames*, which the caller should construct. Our instrumentation transforms:

1. Function call sites to construct context frames
2. Instructions that use static keys to fetch their constant keys
3. Instructions that use dynamic keys to fetch their keys from the context frame

4. All store instructions to encrypt the data

5. All load instructions to decrypt the data

3.3.1 Enabling Context Sensitivity

We seek to support dynamic key assignment for memory instructions that may access multiple equivalence classes depending on their calling contexts. We determine the set of equivalence classes that can be accessed through dynamic keys as follows. For each function in the program, we identify the set of equivalence classes reachable from the function's pointer arguments or pointer return value. From that set, we remove any equivalence classes which contain global variables. If an instruction accesses an equivalence class that contains global variables, then that instruction always accesses that same class, regardless of which context the function is called from. Thus, such an equivalence class can safely be removed from the set. The remaining set of equivalence classes are the *dynamic classes* in that function. Other classes that are used in the function, but that are not in the set (i.e., the classes that were removed because they contain global variables, and the classes that are not reachable from the pointer arguments or pointer return value), are considered *static classes*. During instrumentation, we assign dynamic keys to memory access instructions that target dynamic classes, and static keys to those that target static classes.

Managing Context Frames

We store dynamic keys in *context frames*. For each function that contains instructions with dynamic keys, we first instrument all of the function's callers to create the necessary context frame and to populate the frame with the keys for the actual callee arguments. We then instrument the callee so that instructions accessing dynamic classes read the keys from the context frame.

The context frame contains a mapping from each dynamic equivalence class to that class's encryption key in the current context. To construct context frames we pass the encryption keys as additional positional arguments to the function, and represent the mapping from dynamic equivalence class to key implicitly in the ordering of the context frame.

Recursion

Recursion must be handled with care because it could introduce potentially infinite calling contexts. Recursive function calls can more generally be described as strongly connected components in the call graph of the program. Within the context-sensitive points-to analysis, we handle recursion by detecting strongly connected components in the call graph and collapsing the recursive functions into a single context. This means there is no context sensitivity within a strongly connected component, but the calling context to reach that strongly connected component is still considered. Consequently, recursive functions can still have a context frame, and will then pass that context frame to subsequent recursive calls.

Handling Indirect Calls

Instrumenting indirect call sites complicates context frame management because if care is not taken, different target functions could require different sets of dynamic keys, even if the target functions have the same signature. To correctly instrument indirect call sites we constrain all functions that may be called from the same call site to have the same dynamic classes.

Static Equivalence Classes

Every instruction that accesses a static class will always access *that* static class, regardless of the calling context. Thus, we can safely assign static keys to instructions that access static classes.

Equivalence classes that contain global variables are always *static classes*. To understand why this is always true, consider how a flow-insensitive alias analysis constructs equivalence classes. An alias analysis evaluates all of the instructions in the program and incorporates any aliasing relationship introduced by an instruction into the points-to sets. When a flow-insensitive alias analysis such as ours evaluates a statement such as:

```
void* a = condition ? &global : &function_argument;
```

it will consider pointer `a` an alias for both `global` and `function_argument`, which will therefore be placed into the same equivalence class. This equivalence class will now be a static class, because, no matter which context this function is called from, any instruction that accesses this static class can now potentially access the memory storage location occupied by `global`.

External Code and Data

If a program uses external libraries that cannot be analyzed and instrumented, the code within the library will expect to operate on plaintext data, and will likely crash if given encrypted data. To handle the transition from instrumented code to uninstrumented library code, we follow the same approach used by prior DSR implementations and use wrapper functions. The wrapper function will decrypt all arguments, call the originally intended function, and then encrypt the arguments and any return values. If the program calls a function that we have not implemented a wrapper function for, we cannot encrypt any

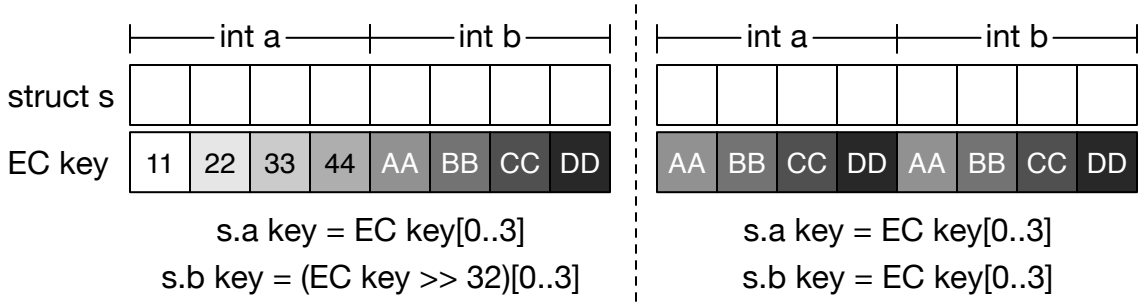

Figure 3.2: Calculating keys for unaligned accesses with our approach (left side) and prior work by Cadar et al. (right side). "EC key" is the key for the equivalence class.

objects in the equivalence classes that may be accessed by that function. To maintain compatibility, such equivalence classes are marked as unencryptable.

3.3.2 Memory Encryption

We instrument memory access operations so that the values are `xor`-encrypted before they are stored to and after they are loaded from memory. The encryption/decryption instructions we add use the unique randomly-generated 8-byte key we assign to their respective target equivalence classes. To use 8-byte keys consistently for all equivalence classes, we must carefully handle memory accesses which are not 8-byte aligned. For example, consider an equivalence class containing a structure with two fields, as shown in Figure 3.2. When accessing field `s.b`, we should shift the key to mask the field's data with the correct part of the key (left side of the figure). Cadar et al.'s DSR implementation assigns weaker, repeating keys (right side of Figure 3.2) to avoid costly shift operations [15].

Our design encrypts all possible equivalence classes. To reduce the run-time overhead, prior DSR systems did not protect equivalence classes that are "safe". An equivalence class is considered safe if a static analysis can show that none of the accesses to that equivalence class can read or write outside the bounds of the target object. This weakens their protection against temporal memory errors such as use-after-free and uninitialized read.

3.3.3 Constructing Equivalence Classes

We use Bottom-up Data Structure Analysis (Bottom-up DSA) [70] to categorize memory objects into equivalence classes. Bottom-up DSA is a context- and field-sensitive points-to analysis that scales well to large programs. It is context-sensitive to arbitrary length acyclic call paths, and it is speculatively field-sensitive. It is field-sensitive for type-safe code, and falls back to field-insensitive for type-unsafe code. The algorithm is unification based and is not flow sensitive.

The output of Bottom-up DSA is a *points-to graph* for each function, which incorporates the aliasing effects of all callees of that function (thus "Bottom-up"). A node in the points-to graph represents a set of memory objects joined through aliasing relationships, and nodes represent disjoint sets of objects. Each node therefore identifies a distinct equivalence class within that function. For each function and its associated points-to graph, we use Bhaktar and Sekar's key assignment algorithm [11] and augment it to differentiate the static and dynamic equivalence classes.

The first step in class assignment is identifying the dynamic equivalence classes. To handle indirect calls, we constrain all possible targets of an indirect call site to have the same dynamic classes. We use Bottom-up DSA to create classes of functions that are all callable from the same call site. The analysis result for these functions is a single points-to graph shared by all functions in the class. Within this graph all arguments and return values for these functions will share the same set of nodes. We use this functionality to compute the set of dynamic classes for all functions in the class simultaneously. We mark all nodes that are reachable from the pointer arguments and the pointer return values of each function in the class, and then remove all nodes that contain global variables or are marked unencryptable. The resulting nodes become the set of dynamic classes for every function in the class. We use the same procedure for functions that are only called directly, but apply the procedure

```
int foo(int *a, int *b)
{
  int *p;
  if (rand() % 2 == 0) {
    p = a;
  } else {
    p = b;
  }
  return *p;
}

int main()
{
  int i, j;
  i = 1;
  j = 2;
  foo(&i, &j);
  return 0;
}
```

Listing 3.2: Operations in callees introduce constraints on equivalence class assignment.

individually to each function.

For each node and its associated equivalence class, we assign a dynamic key if a node is marked as dynamic and a static key otherwise. If a node contains a global variable, we ensure that every such class in all functions uses the same static key. If a node is marked unencryptable, we assign it a null static key which means that memory accesses to this class will not be instrumented.

The equivalence class assignment will be valid for all possible contexts because the points-to graph for each individual function will contain all necessary aliasing constraints introduced by callees. An example of constraints introduced by callees is shown in Listing 3.2. Within the function `foo`, when the pointer `p` is dereferenced, it will have the value of either `a` or `b`. Therefore, `a`, `b`, and `p` will all be in the same equivalence class. Both `a` and `b` are pointer arguments, so they will be in a dynamic equivalence class. We fetch keys to access arguments

```
int foo(int *a, int *b)
{
  return *a + *b;
}

int main()
{
  int i, j;
  i = 1;
  j = 2;
  foo(&i, &i);
  foo(&i, &j);
  return 0;
}
```

Listing 3.3: A program that would lose precision with Top-Down DSA

from the context frame, but in order to use the correct key for `p`, all callers of `foo` must use the same key for both `a` and `b`. Fortunately, Bottom-Up DSA will capture this aliasing relationship and propagate it to callers of `foo`. During the bottom-up inlining phase of Bottom-Up DSA, the graph for `foo` will be inlined into the graph of `main`. At this point, it will merge the nodes for the actual arguments with the formal arguments. This will cause `i` and `j` to be placed in the same equivalence within `main`'s graph.

Along with Bottom-Up DSA, Lattner, Lenharth, and Adve describe the full version of DSA [70]. The complete DSA pointer analysis first performs a bottom-up inlining of the call graph, inlining callees into callers, and then performs a top-down phase where callers are inlined into callees. We use Bottom-Up DSA, which is the result after doing just the bottom-up phase because performing an additional top-down phase would introduce unnecessary restrictions on equivalence class assignment, and reduce the total number of equivalence classes. Listing 3.3 shows an example of a program that will be overly constrained if we were to perform the top-down DSA phase. After the bottom up phase, within `foo`, `a` and `b` will be in separate equivalence classes. If we then perform a top-down inlining, when we inline the graphs for the first call to foo, `a` and `b` will both be merged with the node for `i`. This

```
ssize_t
read_wrapper(int fd, void *buf, size_t count, uint64_t key) {
  ssize_t read_size = read(fd, buf, count);
  if (read_size > 0) {
    xor_memory(buf, read_size, key);
  }
  return read_size;
}
```

Listing 3.4: Wrapper function implementation for `read`.

will cause the arguments to now be placed in the same equivalence class. This constrains our assignment of equivalence classes within `main`. Now, since the arguments to `foo` are in the same equivalence class, we must place `i` and `j` in the same equivalence class as well.

This top-down constraint is not necessary due to our ability to fetch keys from the context frame. For all operations within `foo` (and all callees), we know that the arguments `a` and `b` are independent. Therefore we can assign the keys used for `a` and `b` unique slots within the context frame, and always be able to correctly dereference the pointers using those dynamic keys. If there are callers where the arguments are in the same equivalence class, the same key will simply be duplicated in multiple slots of the context frame. Tracking the calling context implicitly through the keys in the context frame performs the same function as the top-down phase of DSA, but at run-time, giving us greater freedom over equivalence class assignment within each function.

3.3.4 External Code and Data

We handle uninstrumented library code using wrapper functions because the code within the library will expect to operate on plaintext data, and will likely encounter errors if given encrypted data. The wrapper function handles the transition from instrumented code in the protected program to the uninstrumented library code by decrypting any input, and

encrypting any output. A wrapper function has the same signature as the original function, but with additional arguments added to receive the necessary keys through the context-frame. Wrapper functions are written by hand, although the process is very regular, and could be automated.

The implementation of a typical wrapper function is shown in Listing 3.4. This is the wrapper function for `read`, which reads data from a file descriptor. The wrapper takes one additional argument, which is the key for the buffer used for output. In this case since there are no pointer inputs the first operation is to call the original `read` function, and then the returned data is `xor` encrypted with `buf`'s key before the wrapper returns the value returned by `read`.

When interacting with libraries we must also handle aliasing within the library functions. Since the source code of the library function is not available for analysis, we use function summaries that describe any aliasing relationships between the arguments and return values of library functions. This ensures that any aliasing relationships introduced by library functions are correctly captured in the final analysis result.

Special Cases

Wrapper functions are useful for mediating the transition from instrumented code to non-instrumented code, but there are some functions that we cannot use wrapper functions for. These are the functions `setjmp` and `longjmp`, which are used to perform non-local `goto`. The function `setjmp` sets a target and saves the stack context in a `jmp_buf`, and `longjmp` transfers the control flow back to the location where `setjmp` was called. However, `longjmp` requires that the stack context it targets still exists, if the function that called `setjmp` has already returned, that `jmp_buf` can no longer be used as the target of a `longjmp`. If we use a wrapper function, it would call `setjmp`, encrypt the `jmp_buf`, and then return, immediately invalidating the context where `setjmp` was called. To protect `jmp_buf` objects, we inline the

encryption and decryption operations of the `jmp_buf` and leave the stack context unchanged for these functions.

Another special case we handle are `printf` style string formatting functions. These are variadic functions, and the format string determines the number and type of variadic arguments. Bottom-Up DSA handles variadic functions by merging all variadic arguments at a call site into a single node. This is generally necessary since C and C++ place no restrictions on how variadic arguments are used. A wrapper for a variadic function therefore has a single key argument that is used for all variadic arguments. However, for string formatting functions there is no requirement that the arguments alias. Since these functions are very common, and are often called with multiple pointer arguments, we instead perform no merging and allow all variadic arguments to be in different equivalence classes. This means that the context frame is also variable size. However the context frame is normally implemented as positional arguments and therefore must have a fixed size. There is no limit to the number of variadic pointer arguments passed to `printf` so instead of passing keys for the variadic arguments in a fixed layout in the context frame, we pass the key as an additional variadic argument immediately following the pointer. Within the wrapper function we parse the format string, and for any format specifier that causes a pointer-type argument to be fetched, we fetch an additional argument from the argument list and use that as the encryption key.

Unencryptable Data

When we have a wrapper function, we can handle uninstrumented libraries interacting with encrypted data. However, wrapper functions must be implemented for each library function, and if a wrapper function is not implemented, then we are unable to mediate the transition between instrumented and uninstrumented code. To maintain compatibility, we gracefully degrade protection when the set of wrappers is incomplete. If a program calls a library function that does not have a wrapper, the data passed as arguments to, and returned from

the function will not be encrypted. This maintains correctness, but reduces the security because program data is left unprotected. We inform users when this occurs by reporting which library functions are missing wrappers and which equivalence classes this effects, which allows developers to evaluate the security impact of the missing wrapper functions or guide developers to implement the missing ones.

There are also some memory accesses that we cannot safely encrypt even while using wrapper functions. These are accesses to externally defined global variables. Global variables defined in libraries may be accessed by uninstrumented library code at any time, so the protected application cannot encrypt accesses to these variables.

Not encrypting some equivalence classes reduces the security, so we want to ensure that the minimum number of equivalence classes are affected. For each function call in the program, if it targets a library function that does not have a wrapper, then we know we cannot encrypt any memory reachable from the arguments or return value. However we must ensure that the memory is never encrypted in any context, even if the function can receive data from different equivalence classes in different contexts. This puts another constraint on equivalence class assignment that we must propagate bottom-up though the call graph.

To accomplish this we extend Bottom-up DSA. We add metadata to nodes in the points-to graph to indicate if the node cannot be encrypted, and mark nodes that can be accessed by uninstrumented code as unencryptable. Bottom-up DSA represents each function with a separate points-to graph, so first we mark the nodes only in the graph for a single function, then we propagate this information to all relevant contexts. We analyze the statements in the program, if there is a function call to a library function without a wrapper, we perform reachability analysis from its arguments and return value and mark these nodes as unencryptable. This marking is local to the graph of the containing function. We also handle external global variables in a similar fashion. We traverse the collection of global variables in the program, and if a global variable is defined externally, we use reachability analysis to

find all nodes reachable from this global variable and mark these as unencryptable.

After all program statements and global variables have been analyzed, and all local markings made, we propagate the unencryptable markings to all relevant contexts by performing another round of the bottom-up graph inlining step of Bottom-Up DSA. When nodes are merged during graph inlining, we propagate the metadata indicating that a node cannot be encrypted. The final result of this process is the same set of points-to graphs produced by Bottom-UP DSA, but with nodes that escape to unwrapped external code marked unencryptable.

In addition to data, we must also handle function pointers that may escape to external code. An escaping function pointer could be called by the external code without the proper keys in the context frame. Therefore, calls through this pointer must not require dynamic keys. However, it is still possible to pass dynamic keys to direct calls to the same function. To handle this, we maintain two copies of the affected functions—one that accepts dynamic keys and one that does not encrypt accesses to the equivalence classes of the arguments. Note that an attacker may seek to use the version that does not expect encrypted arguments. However, in order to redirect control flow to such a function, the attacker will need to overwrite a code pointer. This memory access will be encrypted, so the attacker will already have to bypass DSR to perform such an overwrite.

3.3.5 Program Transformation

We now present the sequence of operations we perform to apply context-sensitive DSR, and how they are integrated into the build cycle. For DSR it is necessary to have the entire source code of the program available for analysis and transformation, this is done by compiling the program using link time optimization (LTO), and applying context-sensitive DSR at link-time. We first perform all analysis steps, and then use the results of the analysis to guide the

program transformation. The fist step is performing Bottom-Up DSA pointer analysis. We then identify and mark the unencryptable nodes, and perform another bottom-up inlining to propagate the unencryptable marking to all contexts. In the final analysis step we identify the dynamic equivalence classes for each function and determine how many arguments will need to be added.

With the results of all analysis steps available, we then begin our program transformation. First we create a constructor function that will run before `main`. This encrypts the initial values of all global variables with their corresponding keys. Then for each function with dynamic equivalence classes we create new versions of these functions with additional formal arguments to pass the context frame. We then instrument all memory access instructions. Before storing a value, we `xor` it with the key for the destination location, and after loading a value we `xor` it with the key of the source location.

We then instrument the call sites of the program. If there is a call to an external function and we have a wrapper function available for it, we rewrite call instruction to call the wrapper with the correct keys. We follow a similar process for calls to local functions that require dynamic keys. We rewrite the call instruction to call the version of the target function that has additional formal arguments for the context frame, and we pass the necessary keys in these new argument locations. To ensure we pass the correct keys in the correct slot of the context frame we create a mapping of the nodes representing the formal arguments to the nodes representing the actual arguments at that call site. For each node in the context frame, we query this mapping to find the corresponding node in the caller's graph, and add the key associated with this node to the argument list. We also rewrite indirect calls to pass the keys in the context frame, but don't change the target of indirect calls. For indirect function calls, we do not have a specific caller to map to, but all possible targets will share the same graph, so we choose one of the possible target functions and follow the same procedure to construct the context frame.

The final step of the program transformation is changing the targets of indirect calls. At the call site, since the actual target is unknown, we can't simply replace the target with the version that receives a context frame. Instead, for all functions with dynamic keys, we replace all locations where that function's address is taken with the address of the version with additional arguments for the context frame. Since all cases are replaced, any pointer to that function will now refer to the version with dynamic key arguments. This ensures that all indirect calls with dynamic keys will target the function that expects to receive a context frame.

3.4 Implementation

We implemented context-sensitive DSR within LLVM 3.8 [68] using program analyses from the PoolAlloc module [69]. PoolAlloc provides an implementation of Bottom-up DSA, but is unmaintained [39]. We updated PoolAlloc to be compatible with LLVM 3.8 and fixed bugs we encountered during our implementation of context-sensitive DSR. The program analysis and transformation of context-sensitive DSR are implemented as passes over LLVM IR. The analysis uses the results from Bottom-up DSA to determine the number of dynamic keys and identify any equivalence classes that cannot be encrypted as described in section 3.3. We integrate the passes into the LTO plugin, and schedule the context-sensitive DSR passes to be run after all optimization passes. This ensures that the program analysis analyzes the final version of the code, and prevents protections added by context-sensitive DSR from being compromised by later performance optimizations. We implemented wrapper functions for commonly used library functions within the LLVM compiler-rt runtime library. This integrates the wrapper functions into the LLVM system, and makes them available for linking with generated programs. Finally, we added command line options to the compiler driver so that users can easily enable context-sensitive DSR with a single flag, and the compiler driver

will handle passing arguments to the LTO plugin and linking with the wrapper function runtime library.

3.5 Evaluation

We implemented and tested four configurations of DSR to compare how different analysis and design choices affect the performance and precision of a DSR system:

- The **Prior DSR** configuration mimics prior DSR implementations. For this configuration, we implemented a context-insensitive points-to analysis to calculate the equivalence classes, but we did not instrument accesses to safe objects and used weak encryption keys for unaligned accesses (see subsection 3.3.2).
- The **Full Key Size** configuration uses the same analysis, but uses full 8-byte keys for all memory accesses (including unaligned accesses).
- The **Full Context Insensitive** configuration also uses the context-insensitive analysis and 8-byte keys, but encrypts accesses to *all* equivalence classes rather than just the unsafe ones.
- The **Context Sensitive** configuration uses context-sensitive analysis to calculate equivalence classes, 8-byte keys, and encrypts all equivalence classes.

3.5.1 Performance

We measured the run-time overhead of all four configurations using the SPEC CPU 2006 benchmark suite. We ran all benchmarks on system running Ubuntu 16.04 with Linux version 4.15.0 on a 4 core Intel Core i7-3820QM CPU clocked at 2.70GHz with 32KB dedicated L1

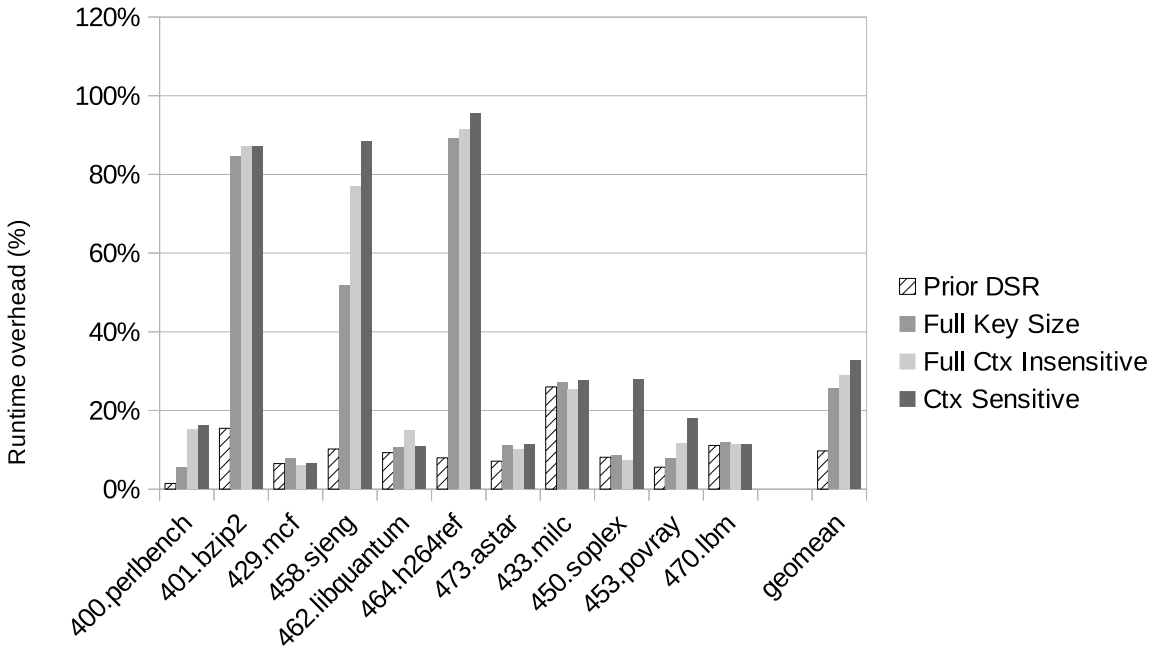

Figure 3.3: Run time overhead on SPEC CPU 2006 benchmarks.

instruction and data caches, 256KB dedicated L2 caches, and an 8MB unified L3 cache, two threads per processor core, and 16GB of main memory.

Figure 3.3 shows the results of compatible SPEC benchmarks for our four configurations. We ran each benchmark three times using the ref input and report the median of the three runs. All configurations of DSR require link-time optimization, so our baseline was also built using link-time optimization. The average run-time overhead we observed for the Prior DSR configuration was 10%, which is consistent with the results reported in prior work [15, 11]. Then, each increasingly secure configuration incurs additional overhead. This is expected because more instrumentation code is needed for the more secure configurations. The average run-time overhead of context-sensitive DSR was 32%, and the highest measured overhead was 95% for `464.h264ref`.

We also measured the overhead of context-sensitive DSR within the context of a webserver. We built the Apache httpd web server with each of the four configurations and compare the throughput to a baseline built with link-time optimization. The server was configured to

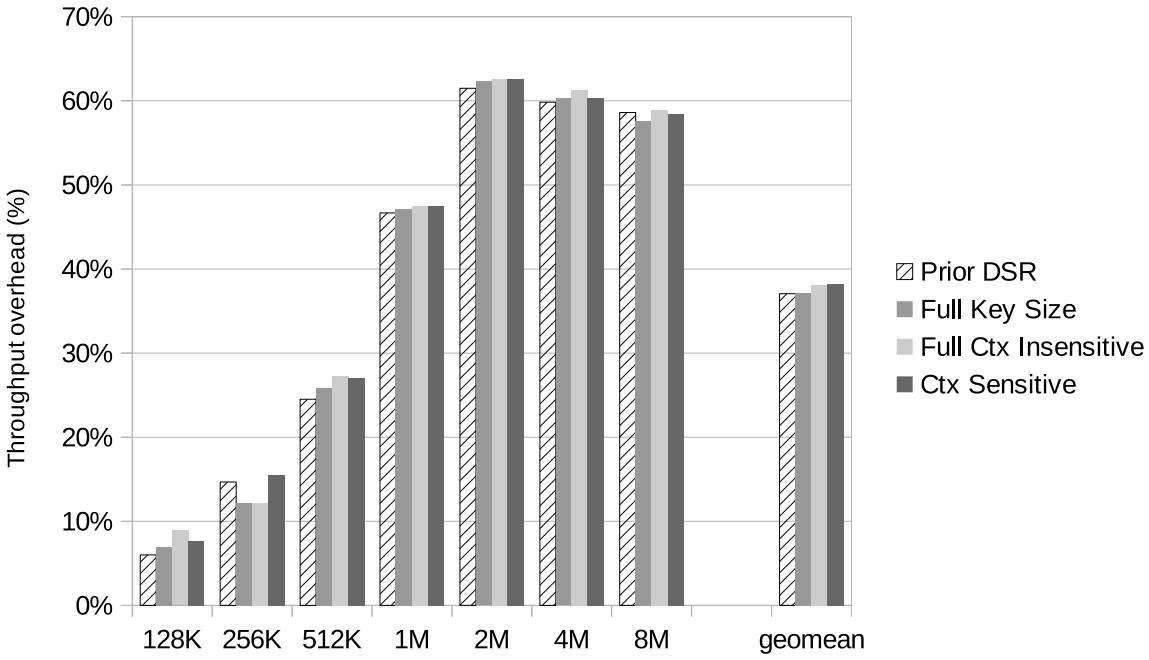

Figure 3.4: Throughput overhead for Apache httpd.

serve static files, and we tested file sizes ranging from 128KB to 8MB and used ApacheBench to generate requests. To minimize the effect of network latency, we generated the requests from the same system running the server. For each file size, we generated 100,000 requests with concurrency level set to four and measured the throughput. We repeated the measurements 30 times and report the average of the 30 runs. Figure 3.4 shows these results. For small file sizes the overhead is less, the throughput overhead for a 128KB file with context-sensitive DSR was 8%. However, the overhead increases dramatically for file sizes 1MB and above, the highest overhead was 63%, which was for 2MB files with context-sensitive DSR. The average throughput overhead across all files sizes for context-sensitive DSR was 38%. There was not a significant difference in the overhead between each of the four configurations, although context-sensitive DSR is able to provide increased precision for Apache httpd (see subsection 3.5.2).

Benchmark	**Prior Work**	**Context Insensitive**	**Context Sensitive**
400.perlbench	313	541 (72.8%)	782 (149.8%)
401.bzip2	86	88 (2.3%)	92 (07.0%)
429.mcf	37	41 (10.8%)	41 (10.8%)
458.sjeng	366	473 (29.2%)	486 (32.8%)
462.libquantum	35	48 (37.1%)	48 (37.1%)
464.h264ref	754	987 (30.9%)	1152 (52.8%)
473.astar	60	63 (5.0%)	89 (48.3%)
433.milc	603	644 (6.8%)	726 (20.4%)
450.soplex	63	68 (7.9%)	142 (125.4%)
453.povray	338	588 (74.0%)	950 (181.1%)
470.lbm	23	25 (8.7%)	25 (08.7%)
precision increase (geomean)		(23.7%)	(52.2%)
nginx	259	357 (37.8%)	575 (122.0%)
ProFTPD	406	628 (54.7%)	667 (64.3%)
sshd	272	358 (31.6%)	559 (105.5%)
WU-FTPD	581	721 (24.1%)	741 (27.5%)
mcrypt	176	222 (26.1%)	257 (46.0%)
Apache httpd	2831	4452 (57.3%)	4606 (62.7%)
precision increase (geomean)		(38.0%)	(68.3%)

Table 3.1: Number of static equivalence classes.

3.5.2 Precision

Context-sensitive DSR can only stop attacks if it can place the legitimate targets of attacker-controlled instructions in different equivalence classes than the memory locations the attacker wishes to access. If an attacker-controlled instruction is used to access a memory location in the same equivalence class as its legitimate targets, the attack will likely succeed. This property also applies to other defenses that rely on static analysis to restrict data flow, including Data-Flow Integrity [17] and WIT [3]. Thus, it is important that the analysis distinguishes memory accesses into as many distinct equivalence classes as possible.

To demonstrate the added security of our context-sensitive analysis, we built the SPEC benchmarks and several programs using three of the four different DSR configurations and we counted the number of encrypted equivalence classes under each configuration. We excluded *Full Key Size* from this comparison, as it uses the exact same equivalence classes as *Prior DSR*. Table 3.1 shows the number of encrypted equivalence classes for each configuration, as well as the percentage increase from the first configuration.

We observe that context-sensitive DSR yields an increased number of equivalence classes compared to prior work and context-insensitive DSR. The loss of precision from a context-insensitive analysis increases the chances that an attacker will manage to find vulnerable code that encrypts data with the desired encryption key. Context-sensitive always yielded the most precise result, although there are cases where both context-insensitive and context-sensitive produced the same number of equivalence classes. This was seen for the benchmarks `429.mcf`, `462.libquantum`, and `470.lbm`. The precision of different analysis algorithms depends on the structure of the program, for these programs a context-insensitive analysis provides an equally precise result as a context-sensitive analysis. In general, context-sensitive provides a benefit, on average the number of equivalence classes increased by 52.2% for SPEC benchmarks and 62.7% for the programs we evaluated. It is important to note that

Benchmark	Context Insensitive		Context Sensitive	
	Average	Maximum	Average	Maximum
400.perlbench	3.36	3802	3.21	3816
401.bzip2	1.09	9	1.04	9
429.mcf	1	3	1	3
458.sjeng	1.18	36	1.15	36
462.libquantum	0.82	1	0.81	1
464.h264ref	1.38	150	1.04	150
473.astar	1.97	41	1.46	41
433.milc	0.91	43	0.79	43
450.soplex	1.25	267	0.61	267
453.povray	6.47	3125	34.79	3125
470.lbm	1	2	1	2
nginx	3.58	2059	3.67	1956
ProFTPD	1.43	1579	1.26	1054
sshd	1.97	331	1.18	201
WU-FTPD	1.39	491	1.12	303
mcrypt	0.95	127	0.84	116
Apache httpd	1.35	4045	1.21	3131

Table 3.2: Number of allocations per static equivalence class.

the additional equivalence classes identified and protected by context-sensitive DSR also include memory that is considered safe and thus left unencrypted by prior work (see subsection 3.3.2). This gives context-sensitive DSR additional resistance against temporal memory vulnerabilities such as use-after-free or uninitialized-read.

Another important security property is the size of the equivalence classes, since the larger an equivalence class gets, the easier it generally becomes to illegitimately access variables within that class. To quantify equivalence class sizes, we modified our analyses to track the number of allocation sites (global, stack, and heap) contained within an equivalence class. For global and stack allocations, these correspond to variable declarations, for heap allocations they are calls to heap allocator functions like `malloc`. We counted both the average and maximum number of allocation sites per equivalence class, as shown in Table 3.2. The results show that, in general, the context-sensitive analysis used by context-sensitive DSR

gives lower number of allocation sites across the benchmarks, and in many cases reduces the size of the largest equivalence class substantially. Note that some benchmarks actually show an increase in average number of allocation sites. This is because an allocation site can be counted multiple times in different contexts with context-sensitive analysis. This is evident in the results for `453.povray`, for which the average size of an equivalence class increased substantially. Context-sensitive DSR still provides improved security, the results in Table 3.1 show that the number of equivalence classes increased by 181.1%. While the size of individual equivalence classes in `453.povray` did not change, context-sensitive DSR placed many allocation sites in different equivalence classes for different contexts, making it more difficult for an attacker to perform a successful overwrite compared to context-insensitive DSR.

3.5.3 Real World Exploit

We evaluated context-sensitive DSR against a recent data-oriented attack presented by Hu et al. [56]. The attack exploits a format string vulnerability in the *wu-ftpd* server to perform privilege escalation. Specifically, the attack overwrites a global pointer to a `struct passwd`. The overwritten pointer is later read and then dereferenced by the server, and the dereferenced value is interpreted as a user ID. This user ID is subsequently used as an argument for a `setuid` call. The attacker escalates the privileges of the vulnerable application by overwriting the global pointer with the address of memory that contains the value 0, which is the user ID of the root user.

We built two versions of the *wu-ftpd* binary: a base and a version protected by context-sensitive DSR. We then tested the exploit against both versions. The exploit was successful against the base version, but did not work against the protected version. While the attacker is still able to overwrite the pointer with context-sensitive DSR, the subsequent read used a

different encryption key than the instruction that overwrote the pointer, making it impossible for the attacker to reliably control the outcome of the overwrite. This causes the argument to the `setuid` call to be an unpredictable value. Context-sensitive DSR identifies three equivalence classes involved in this exploit: the class accessed by the vulnerable instruction during valid executions, the class of the pointer variable, and the class used for dereferences of the pointer. These classes are accessed using distinct keys, k_v, k_p, and k_d respectively. To reliably control the result of this exploit an attacker would have to guess two 64-bit secret values, $k_v \oplus k_p$ and k_d, and therefore the attack has a low chance of succeeding.

3.6 Discussion

We have improved an important aspect of DSR, namely the number of attacks that will be stopped. This is accomplished by increasing the precision of the static analysis. We also take a principled approach in our implementation and do not make design decisions that compromise security such as omitting any equivalence classes from instrumentation or reducing the width of keys in the presence of unaligned memory accesses. While previous implementations did make these compromises [15, 11], they were aware of the security implications of doing so, and provided justification for their design choices. However, while we increase the precision, there are still limitations that context-sensitive DSR shares with prior work.

Chief among these is the weakness to known-plaintext attacks. If an attacker knows the value of the plaintext data, and is able to leak the encrypted data, they can deduce the key because the encryption is implemented simply as an XOR operation. This can be mitigated by randomizing the layout of the program and data, so an attacker cannot know the intended plaintext value of any piece of data. We expect that context-sensitive DSR will be deployed on systems with ASLR enabled, and further mitigation could be provided by more fine-grained forms of randomization, for example data structure layout randomization [72].

A related issue is that randomizing data using `xor` operations does not provide any integrity checking. This gives the attacker leeway to exchange encrypted data within the same equivalence class without knowing the key. In order to craft an exploit using this technique, the attacker will still need to know the meaning of the encrypted data, although they do not need to know the exact plaintext value. This is analogous to the limitation of many CFI approaches where an adversary can swap a pointer with another pointer as long as both pointers are allowed targets for a given indirect branch. The lack of integrity checking is an example of a performance-security trade off, and like CFI, DSR makes attacks substantially harder to construct.

Another attack vector against DSR is to target variables for which the range of valid values is a small subset of the possible values for the data type. An example is Boolean variables in C programs. A memory byte representing a Boolean value can have 2^8 different values, but only one of them will be interpreted as `false`. If an attacker wishes to change a `false` value to `true`, the attack will have a high probability of succeeding. In practice, many C programs are written such that Boolean variables will only have a limited number of values, often just 0 or 1. Attacks targeting these values could be mitigated by using a range analysis to identify the valid ranges and inserting checks to ensure the plaintext data is always within the allowed range.

A possible way to bypass DSR is to leak the encryption keys directly by disclosing the code. In our system, the keys will appear as constant operands to instructions in the machine code of the program. If an attacker can leak the code, they can discover every key used by the program. DSR provides mitigation against memory leaks, the leaked data will be encrypted with the key assigned to the instruction used to perform the leak. However, if an attacker can determine that key, they can then leak the code and discover every other key used by the program. This attack vector requires the attacker to read the code, so it can be stopped by enforcing execute-only code. Execute-only code cannot be enforced

using memory permissions on x86, but there are systems like Readactor [26] that provide enforcement of execute-only code.

Context-sensitive DSR also introduces a new potential issue. With context-sensitive DSR, keys are not always constants, instead they can change depending on the calling context. This means that an attacker may be able to alter the keys. The context frame is implemented by passing additional function arguments which contain dynamic keys. Under the calling conventions of the System V ABI [78], used by systems running Linux on the AMD64 architecture, the first 6 integer arguments are passed in registers, and the remaining arguments are passed in the stack. Data in registers is not vulnerable to memory corruption, but depending on the number of arguments, keys may be placed on the stack where they are vulnerable to overwrites. Even if the dynamic key arguments are placed in registers, the compiler may choose to save them on the stack to free the register for other operations, and register values will often need to be saved on the stack during function calls. These factors mean it is likely that there are some points in a program where keys will appear in writable memory, and are vulnerable to tampering. It has been shown that CFI implementations that inadvertently place critical data onto the stack can be bypassed by overwriting that data [24], and an attacker may be able to similarly bypass context-sensitive DSR. DSR mitigates overwrites by encrypting data, so an attacker that does not know any keys will not be able to control the result of an overwrite. Even so, this changes the invariant that keys are immutable values within the program, and adds an additional attack vector.

Keys that will later be used as part of an encryption or decryption operation will only appear on the stack, so they can be protected using general purpose stack protections. For example stack canaries [25] can be placed on the stack adjacent to saved keys to indicate if an overwrite has occurred. Another option is to deploy context-sensitive DSR with Safe Stack [65]. Safe Stack separates the stack into a two stacks, an unsafe stack containing objects that are vulnerable to overflows, and a safe stack containing objects that are always

accessed safely. Encryption keys will be placed on the safe stack since they are placed on the stack only by the compiler.

Another drawback of context-sensitive DSR is the increased performance overhead. While it may be acceptable for many applications, it is still higher than prior DSR systems. While it allows for more precise analysis, tracking context adds additional overhead, and we also make different design decisions to prioritize security over performance. In many exploit mitigations there is the concept of a trade-off between performance and security, and we feel that the increased security guarantees provided by context-sensitive DSR justify the additional overhead.

3.7 Conclusion

Memory corruption exploits are dangerous threats to secure software. CFI and ASLR are effective mitigations against code reuse attacks, but they do not stop non-control data attacks. DSR is a promising approach to stop both non-control and control data attacks, however existing work used imprecise analysis which may allow sufficient leeway for an attacker to construct an exploit. Additionally, in order to reduce run time overheads, prior implementations chose to compromise the security level by reducing key size for unaligned memory accesses and only protecting equivalence classes that are vulnerable to spatial memory errors.

We presented context-sensitive DSR, an improved version of DSR that has higher precision. To achieve the increased precision we construct equivalence classes using context-sensitive points-to analysis and dynamically track run-time context using context frames. The higher precision increases the number of attacks context-sensitive DSR will stop because it is more likely that an attacker will introduce forbidden data flow during an attack.

Context-sensitive DSR also provides increased security by always using 8-byte encryption

keys and encrypting all possible equivalence classes. This makes context-sensitive DSR harder to bypass. It is unlikely an attacker will correctly guess a random 8-byte value, and encrypting all equivalence classes provides additional protection against temporal memory errors which would otherwise be missed.

Context-sensitive DSR does come with higher run time overheads, however it provides protection against a much wider range of attacks than prior DSR systems. Although the overheads are increased, they still compare favorably with systems that enforce spatial and temporal memory safety [82].

Chapter 4

Hardware Assisted Randomization of Data

4.1 Motivation

Context-sensitive DSR improves the precision available to DSR systems, but fails to address some of the other limitations discussed in section 3.6. However we can protect mutable keys and reduce the performance overhead by enhancing context-sensitive DSR to utilize a hardware extension designed to support the instrumentation operations of DSR. This motivated our work on hardware assisted randomization of data (HARD). A hardware extension protect keys from being leaked or tampered with, and can substantially reduce the run time overhead by accelerating the encryption operations. The hardware extension will manage the keys used by the program and provide specialized instructions for the encryption operations used by DSR. Belleville et al. [10] present one such hardware extension which can be used to implement a DSR system. We utilized this hardware to create HARD, a hardware-assisted, context-sensitive, DSR system that is resilient against key leakage and

tampering, and has low run time overhead.

4.2 Design

Context-sensitive DSR requires four general operations:

1. Load a value from memory and XOR with key.
2. XOR a value with key and store to memory.
3. Fetch a key from the current context frame.
4. Construct a context frame.

A software-only implementation of context-sensitive DSR uses standard `xor` instructions for encryption and decryption operations, constructs context frames by passing additional arguments to functions, and accesses keys in the context frame as arguments at a specific offset within the argument list. Much of the complexity required for context-sensitive DSR is in the static analysis, the instrumentation operations added during the program transformation are straightforward, and can be implemented directly in hardware. To support these operations we designed a hardware extension that can be used to implement DSR systems [10]. The hardware extension provides instructions that are able to perform all of the four operations necessary for context-sensitive DSR. Specifically the hardware is able to accelerate the encryption and decryption operations, and manages encryption keys and context frames to ensure they cannot be leaked or tampered with. We use the same static analysis described in section 3.3 to construct HARD, a system to perform hardware assisted randomization of data. HARD provides the same increase in precision provided by context-sensitive DSR, it is resistant to attacks that attempt to bypass DSR by leaking or tampering with keys, and it has lower performance overheads.

4.2.1 Hardware Capabilities

Our hardware extension extends the RISC-V instruction set architecture [105] by providing two new sets of instructions. One is used to load data from, or store data to encrypted memory. The other set is used to construct and manage context frames.

For each type of load and store instruction in the RISC-V instruction set architecture, they provide a variant to access encrypted memory that decrypts data when loading and encrypts data when storing. The key is specified as an immediate operand to the instruction, but instead of providing the key value directly, the operand is an identifier the hardware uses to fetch the actual key. The actual key values, and the memory used to store them, are managed by the operating system kernel and the hardware. The operating system never makes the memory containing the keys available in the address space of the protected process. These enhanced load and store instructions allow us to efficiently perform operations 1 and 2 from the above list.

The specialized load and store instructions are also used to perform operation 3. One bit within the immediate operand to these instructions is used to specify whether the key identifier is interpreted as a global identifier of a static key, or as the index into the context frame of a dynamic key. If this bit is set, the hardware will transparently fetch the key from the context frame and use that key for the encryption operation.

The second set of instructions they provide is for constructing and managing context frames. The hardware provides a stack abstraction for context frames, referred to as the Context Stack, and provides instructions to push and pop entries onto the stack. There are always two context frames that are accessible, there is the currently active frame, from which keys can be fetched, and there is the under construction frame, to which keys can be copied. A context frame is constructed by using new instructions which copy keys to the under construction frame. The key copied to the under construction frame can either be a static

key, identified by it's unique identifier, or a dynamic key, identified by an offset in the currently active context frame. Once a context frame has been constructed, the `drpush` instruction pushes the under construction frame onto the stack, and makes it the currently active frame. The argument to `drpush` is the number of slots in the currently active context frame. This is necessary because the hardware only provides the ability to construct and access the frames, but tracking context, ensuring the correct keys are placed in the context frame, and determining the size of each context frame must be handled at the software level. When a function returns, the `drpop` instruction is used to deactivate a context frame. It pops a context frame from the stack, deactivating the active context frame and activating the previous context frame. This set of instructions allows us to implement operation 4. The stack used to hold context frames is managed by the hardware and operating system kernel, and is separate from the program's stack. It can only be accessed through this set of instructions, and the memory used for the stack is never mapped within the address space of the protected process.

4.2.2 Software

The additional hardware provides mechanisms to efficiently perform the operations needed for a DSR system, but it does not identify equivalence classes or automatically track context. Much of the complication is still handled at the software level during compilation. We reuse the analysis and equivalence class identification described in section 3.3. The only change we must make is how we handle string formatting functions. For `printf` style functions, we had previously extended the context frame by adding additional variadic arguments for the keys of variadic pointer arguments. We can no longer treat these functions as a special case since the hardware requires all context frames to be fixed size. Instead, we handle `printf` style functions the same as other variadic functions, all variadic arguments are placed in a single equivalence class, and a single slot is allocated in the context frame to hold this key. This

does reduce the precision compared to software-only DSR, but the improved performance and resilience against key leakage that the hardware provide justify this trade-off of precision for this special case.

The HARD program transformation performs the same analysis and instrumentation as context-sensitive DSR, but we modify it to target the new hardware. We instrument load and store instructions by first marking them with the identifier of the key they will use, either a static identifier for a static equivalence class, or an offset in the context frame for a dynamic equivalence class. Then we rewrite these instructions to replace them with the new load and store instructions that access encrypted data and provide the key identifier as the immediate operand.

We instrument function calls to track calling context using the hardware instructions for constructing and activating context frames. Before function calls that require dynamic keys, we insert instructions to construct a context frame. Then immediately before the call we ensure the frame is activated by inserting `drpush`. We track how many keys are in each context frame, and provide this as the size argument to `drpush`. It is important to note that we need to insert `drpush` before all calls, even if the target function does not require dynamic keys. Although they do not require anything to be copied into the context frame, they may call a function that does. We must activate an empty context frame before calling these functions because `drpush` requires the size of the currently active context frame. If we do not do this, the size of the currently active frame would depend on the caller of the function, which will lead to errors because we could not statically provide the correct size argument to `drpush`.

To restore the correct context we insert `drpop` before returns. We add the instruction before all return instructions in functions that require a non-empty context frame. However, we cannot safely do the same within functions that do not require dynamic keys. These functions require an empty context frame to be created, but it is possible that a function

with no dynamic keys is called indirectly from a call site that also targets a library function. Before the call we insert instructions to construct an empty context frame, but external code will never execute `drpop`. To ensure that the correct context is always restored in these cases, we insert the `drpop` instruction immediately after the call instruction. This is only required for functions that have empty context frames. If the context frame is non-empty, the call site is either targeting an internal function, or a wrapper function for a library function, and in both cases we will insert `drpop` before the return instruction.

4.3 Implementation

We implemented HARD within LLVM 3.8 for RISC-V. Our implementation is based on our implementation of context-sensitive DSR and we reused portions of the Context-Sensitive DSR analysis and transformation. In order to target the new hardware, we extended LLVM to be able to emit the new instructions. We add intrinsic instructions to LLVM IR that represent the new hardware instructions, and modified the context-sensitive DSR transformation to insert these IR instructions. We also modified the lower level representations LLVM uses to transform IR into machine code to represent these instructions, and added support withing the code generator to emit the new instructions.

When using the hardware, it is no longer necessary to assign a specific key value to each equivalence class, instead the key is specified by an identifier. Instead of assigning a key that is used in encryption operations, we change the transformation to assign these identifiers and supply them as the operands to the hardware instructions. However, encoding details of the hardware instructions limit the total number of keys available to the program. The hardware limits the size of static IDs to eleven bits and dynamic IDs to nine bits. This means that the maximum number of keys available to the program is 2048, and no context frame can exceed 512 entries. No program we evaluated had a context frame exceeding 512

entries, and while most programs we evaluated used less than 2048 keys, some programs did use more. We assign the IDs consecutively from the available values, and if we exceed 2048 we will reuse IDs, and therefore keys, between multiple equivalence classes. This reduces the effective number of equivalence classes to 2048, we discuss the security impact of reusing keys between equivalence classes in subsection 4.4.2.

The transition to uninstrumented library code is still handled using wrapper functions, however we must ensure that the wrapper functions also use the new hardware capabilities. We modified the wrapper functions used by context-sensitive DSR to use the added instructions. The wrapper functions are written in C, and we used inline assembly code to insert the new instructions used to perform encryption and decryption operations. We made heavy use of C preprocessor macros to provide an abstract programming interface to implement wrapper functions. The macro interface is used to perform the memory encryption operations, and the macro definitions control the code that is inserted. We use different definitions of the preprocessor macros depending on if we are targeting hardware-assisted, or software-only DSR, which allowed us to share the same wrapper function implementations between HARD and software-only context-sensitive DSR.

4.4 Evaluation

We implemented and tested several configurations of HARD's analysis and instrumentation passes and compared them to prior DSR implementations, these are the same configurations presented in section 3.5. We evaluated the different configurations on RISC-V, both with and without utilizing hardware support:

- The **Prior DSR** configuration mimics prior DSR implementations. For this configuration, we used a context-insensitive points-to analysis to calculate the equivalence

Benchmark	Prior DSR	Full Key Size		Context Insensitive		Context Sensitive	
	SW Only	SW Only	HW Supp.	SW Only	HW Supp.	SW Only	HARD
164.gzip	11.42%	40.17%	3.19%	70.63%	4.43%	70.94%	7.68%
175.vpr	20.14%	40.29%	8.67%	51.24%	9.64%	51.57%	9.81%
176.gcc	12.35%	22.43%	3.23%	29.00%	3.93%	34.68%	6.37%
181.mcf	7.91%	7.88%	3.70%	7.80%	3.74%	7.85%	3.69%
186.crafty	35.61%	58.81%	6.77%	68.20%	7.03%	70.83%	8.04%
197.parser	3.59%	7.21%	0.43%	17.97%	0.87%	25.17%	4.70%
252.eon	10.85%	17.51%	6.18%	18.21%	5.55%	22.59%	8.88%
253.perlbmk	1.65%	1.58%	0.46%	22.22%	1.35%	23.19%	1.11%
254.gap	14.69%	14.20%	5.75%	21.48%	6.32%	24.26%	6.64%
255.vortex	11.95%	28.32%	2.58%	28.75%	4.33%	43.68%	12.33%
256.bzip2	8.52%	76.04%	5.17%	83.92%	6.81%	83.98%	5.78%
300.twolf	16.11%	29.11%	3.51%	48.43%	4.47%	54.13%	4.70%
geomean	12.60%	26.99%	**4.11%**	36.96%	**4.85%**	40.96%	**6.61%**

Table 4.1: Run-time overhead of HARD and software-only DSR on SPEC CINT 2000.

classes, but we did not instrument accesses to safe objects and used weak encryption keys for unaligned accesses (cf. subsection 3.3.2).

- The **Full Key Size** configuration uses the same analysis, but uses full 8-byte keys for all memory accesses (including unaligned accesses).

- The **Full Context Insensitive** configuration also uses the context-insensitive analysis and 8-byte keys, but encrypts accesses to *all* equivalence classes rather than just the unsafe ones.

- The **Context Sensitive** configuration uses context-sensitive analysis to calculate equivalence classes, 8-byte keys, and encrypts all equivalence classes.

4.4.1 Performance

To measure the run-time overhead of HARD we tested the four different configurations with and without architectural support. We were unable to utilize hardware support for the Prior DSR configuration because the hardware does not support variable sized keys, for this

configuration we only present software-only results. We evaluated all configurations on an FPGA implementation of RISC-V with our hardware extension. The FPGA has a 25MHz clock and has 256MiB of DDR3 memory. We ran the RISC-V port of Linux 4.1.17, which we modified to manage the additional hardware. Since this platform is severely resource constrained, we opted to evaluate the run-time performance using the SPEC CINT 2000 benchmark suite, instead of the more recent SPEC CPU 2006. For the same reason, we also ran the benchmark programs on the *train* inputs, as the board does not have enough memory to use the *ref* inputs.

Table 4.1 shows our evaluation results. Each increasingly secure configuration results in additional overhead, and in all cases hardware support is able to substantially reduce the overhead. The overhead of the most precise configuration is 6.61% with hardware support, while the overhead of the software-only implementation is 40.96%. Increasing the security still increases the overhead, however by making efficient use of additional hardware, HARD is able to provide the same level of protection as context-sensitive DSR with about one sixth the performance overhead. Additionally, HARD provides greater security with lower performance overhead than a software-only implementation of less secure DSR schemes like those described in prior work [11, 15].

4.4.2 Security

HARD also has desirable security properties. The additional hardware provides HARD resilience against key disclosure and tampering. The hardware and operating system manage the memory used to store encryption keys and context frames and these cannot be accessed within the protected process. The processor accesses these memory regions directly using the physical memory address, the operating system allocates the memory regions, and that memory is never mapped to the virtual memory range of any user mode process. This

Benchmark	Prior Work	Context Insensitive	Context Sensitive
164.gzip	77	127 (64.9%)	145 (88.3%)
175.vpr	630	717 (13.8%)	801 (27.1%)
176.gcc	1221	2115 (73.2%)	2957 (142.2%)
181.mcf	37	41 (10.8%)	41 (10.8%)
186.crafty	943	1133 (20.2%)	1161 (23.1%)
197.parser	289	343 (18.7%)	443 (53.3%)
252.eon	1160	1556 (34.1%)	1722 (48.5%)
253.perlbmk	268	491 (83.2%)	528 (97.0%)
254.gap	196	394 (101.0%)	499 (154.6%)
255.vortex	763	911 (19.4%)	1598 (109.4%)
256.bzip2	71	96 (35.2%)	106 (49.3%)
300.twolf	442	692 (56.6%)	797 (80.3%)
precision increase		(41.4%)	(68.1%)

Table 4.2: The number of static equivalence classes that each analysis finds for SPEC CINT 2000 benchmarks.

memory can only be accessed by the specialized memory encryption and context management instructions, so an attacker cannot disclose the encryption keys or tamper with context frames. This protects mutable dynamic keys, and makes HARD more difficult to bypass than a software-only DSR system.

Since HARD uses the same analysis as context-sensitive DSR, it also provides increased precision compared to prior DSR work. We fully analyze the benefits of increased precision for context-sensitive DSR in subsection 3.5.2. The same results are valid for the equivalence classes used by HARD. We also present additional results for the specific benchmarks used to evaluate HARD. For the SPEC CINT 2000 benchmarks, Table 4.2 shows the number of static equivalence classes and Table 4.3 shows the number of allocation sites per equivalence class. From this set, there is one benchmark, 181.mcf, where the context-insensitive and context-sensitive analysis find the same number of equivalence classes. There are programs where a context-insensitive algorithm will produce an equally precise result as a context-sensitive one, but the context-sensitive analysis used by HARD and context-sensitive DSR consistently provides the most precise result.

Benchmark	Context Insensitive		Context Sensitive	
	Average	Maximum	Average	Maximum
164.gzip	1.21	8	1.09	8
175.vpr	1.21	71	1.13	48
176.gcc	2.48	2824	1.87	2187
181.mcf	1.07	3	1.07	3
186.crafty	1.11	57	1.08	42
197.parser	1.73	379	1.41	290
252.eon	1.47	519	1.23	273
253.perlbmk	4.13	1875	4.24	1872
254.gap	4.18	1355	3.73	1270
255.vortex	2.99	1521	3.73	1071
256.bzip2	1.11	11	1.01	3
300.twolf	1.05	18	0.91	9

Table 4.3: Number of allocations per equivalence class for SPEC CINT 2000 benchmarks.

The hardware extension used by HARD limits the number of static and dynamic keys that can be used. The program cannot use more than 2048 static keys, and no function can use more than 512 dynamic keys. The limit on dynamic keys is sufficient for the programs we evaluated, the maximum number of dynamic keys needed was 13. However, our experience shows that there are programs that may use more than 2048 static keys, and the increased precision of context-sensitive analysis can make this more likely. Of the SPEC CINT 2000 benchmarks we used to evaluate HARD, only one, 176.gcc, had more than 2048 static equivalence classes. When using the hardware to protect 176.gcc, we will have to reuse equivalence class IDs and therefore encryption keys. Other techniques that have a space constraint imposed on the protection mechanism are also limited in the protection they can provide. For example, the entries in the color table used by WIT [3] are 1-byte long, which limits WIT to use 256 distinct colors at most. HARD's limit of 2048 IDs allows it to protect much more complex programs than WIT. The security impact of static ID reuse could be reduced by carefully choosing which equivalence classes may share IDs.

4.5 Conclusion

DSR is an effective defense against a wide range of memory corruption attacks, and context-sensitive DSR is the most complete form of DSR currently available. However context-sensitive DSR fails to protect the encryption keys from tampering or disclosure, and the increased protection it provides comes with higher run time overheads, which may be unacceptable for some applications.

HARD is a hardware assisted implementation of context-sensitive DSR. We make efficient use of an extension to the RISC-V ISA designed to support common operations needed by DSR systems. By making effective use of the hardware, HARD provides the same increased precision of context-sensitive DSR, but with much lower run time overheads. Additionally, we use the hardware to manage the encryption keys and protect them from attacks. On systems with this hardware support, HARD is able to provide precise protection against spatial and temporal memory errors, with low run time overheads.

Chapter 5

Kernel Address Leak Detector

5.1 Motivation

Although much progress has been made on techniques that mitigate memory corruption, many techniques have failed to see widespread deployment due to performance concerns [99]. Randomization-based defenses are among the techniques that *have* found or that are finding their way into commodity systems, thanks in no small part to their relatively low run-time performance impact.

Since memory corruption exploits typically require knowledge of the memory layout, randomization-based defenses force adversaries to expend additional effort to learn the memory layout before attempting to take control of the victim process. In the simplest case, the adversary seeks to overwrite a specific variable in the program. With randomization, the attacker must first guess or discover the address in memory used to store that variable. If the adversary resorts to guessing, odds are that the program will crash and thus alert defenders before a correct guess is made. Since randomization relies on hiding the memory layout, adversaries look for ways to leak the memory contents as an alternative to raw

guessing [97, 90].

Randomization can be applied at many granularities and at different stages in a program's development and deployment cycle [67]. Address space layout randomization (ASLR) is the de facto standard way to add randomness to the memory layout of a running process [85]. ASLR shifts the base address of each segment (code, data, stacks) by adding a random offset to its default location. Since ASLR preserves the internal structure of each segment, an adversary that leaks a single pointer to a section can infer the exact layout of that entire section.

ASLR was first adopted for code running in user mode, but is now commonly applied to operating system kernels as well. The Linux kernel, which is used for desktop PCs, servers, and mobile phones (as part of the Android operating system), now supports kernel ASLR (KASLR). KASLR functions similarly to ordinary ASLR for user-space programs. During system boot, the locations of code and data are placed at a randomized offset within the available kernel memory space.

KASLR independently randomizes the base addresses of the kernel, stack, and heap at boot time, but it does not randomize the internal layout of any of the regions. Furthermore, KASLR uses the same randomization offset for the code and global data within the kernel region. Attackers can therefore infer the entire layout of the kernel, including that of its executable code, if they can discover a single pointer to a known instruction or global data variable. Since pointers to kernel code or global data must be kept secret, we refer to such pointers as **sensitive** pointers.

If attackers discover a sensitive pointer, and thus learn the kernel code layout, they could then attempt to mount privilege escalation attacks against the kernel using either code reuse or non-control data corruption. A Google Project Zero member recently demonstrated a return-oriented programming attack against Android smartphone kernels by exploiting

overflow bugs in a touch screen driver [46]. At BlackHat 2017, a security researcher presented a privilege escalation attack on Android by combining a jump-oriented programming attack to get a root process and a data-only attack to disable SE-Android [92].

The first step in both of these attacks is leaking a kernel pointer. A number of recent vulnerabilities show that it is feasible to leak such pointers to non-privileged applications running in user-space. These vulnerabilities exist in two forms: those that exploit memory corruption to disclose pointers [92, 29, 28, 31, 30, 27], and those where the kernel code directly discloses addresses [47, 32]. There are also attacks that infer kernel-space addresses indirectly through micro-architectural side channels [62, 58, 49]. While micro-architectural side channels can be prevented using kernel page table isolation [48], and there are many strategies to statically detect memory corruption vulnerabilities within the kernel [76, 45, 75], there are few mitigations that prevent or detect direct disclosure vulnerabilities.

In this section, we present the Kernel Address Leak Detector (`KALD`), a static analysis tool that can find locations where the kernel directly leaks sensitive addresses to user-space memory. Contrary to the current practice of labor-intensive and error-prone manual code reviewing, our tool can automatically analyze the kernel source code to find these issues.

5.2 Background

5.2.1 Code-reuse Attacks

Commodity operating systems primarily rely on enforcement-based techniques to prevent certain types of exploits. Enforcing strict data execution prevention (W$\oplus$X) prevents most code-injection attacks [6], as pages cannot be simultaneously writable and executable. Modern operating systems also prevent executing user-space code with kernel privileges by using

supervisor mode execution prevention on x86 [61] and privilege execute-never on ARM [7]. Unfortunately, attackers can circumvent these mitigations by crafting exploits that reuse existing kernel code [107, 43]. Code-reuse attack techniques exploit memory corruption vulnerabilities to alter control flow data such that existing sequences of instructions (a.k.a. *gadgets*) are chained together in order to perform malicious actions. These techniques include return-oriented programming (ROP) [91], which targets return addresses on the stack, and jump-oriented programming (JOP) [18], which targets function pointers.

5.2.2 Kernel ASLR

One common defense against code-reuse attacks is address space layout randomization (ASLR). ASLR introduces randomness into the memory layout of the program. For example, user-space ASLR on Linux randomizes the location of stack, mmap, brk and text sections. Similar to user-space ASLR, KASLR is implemented in modern operating systems by adding a random offset, the **KASLR offset**, to the default loading address of kernel, so that the location of the kernel's code and global data are randomized during system start up. This stops kernel code-reuse attacks because an attacker no longer knows the exact address of any gadgets. An attacker will now have to guess the location of the gadgets, and if the layout entropy is large enough, it is very likely that the guessed locations will be wrong, and the attack will fail.

User and kernel ASLR are effective countermeasures to code-reuse attacks and are widely adopted by mainstream operating systems [63], as shown in Table 5.1.

Table 5.1: ASLR adoption in mainstream operating systems

Operating System	**ASLR Type**	**Year**
Linux 2.6.12	User	2005
Windows Vista	User/Kernel	2007
OS X 10.5	User	2007
iOS 5	User	2011
Android 4.0	User	2011
OS X 10.8	Kernel	2012
iOS 6	Kernel	2012
Linux 3.14	Kernel	2014
Samsung Android 6.0	Kernel	2016
Android 8.0	Kernel	2017

5.2.3 Address Leakage

The most common technique to bypass ASLR is code derandomization using address leakage. When KASLR is applied, the first step towards a successful code-reuse attack is to bypass KASLR by leaking a code or global data pointer [92].

There are several ways that an attacker can find such a pointer. One option is to exploit a memory disclosure vulnerability to read a kernel address directly from the kernel memory. However, this requires finding a reliable memory disclosure, which may not be available. Another option is to find a case where the kernel writes an address to a user-readable location. This could be user-space memory, if the kernel writes the value of a kernel pointer there as the result of a system call, or it could be a user-readable log file, if the kernel outputs addresses as part of log messages as in recently discovered vulnerabilities [47, 32]. Most of the kernel's code was written before KASLR was implemented, and there are cases where the kernel discloses addresses to user processes. As a result, it is often possible to find a kernel address. Kernel developers now know that kernel addresses should not be written to user-space memory or logged, but without an effective way to catch these issues, there may still exist cases where addresses are written.

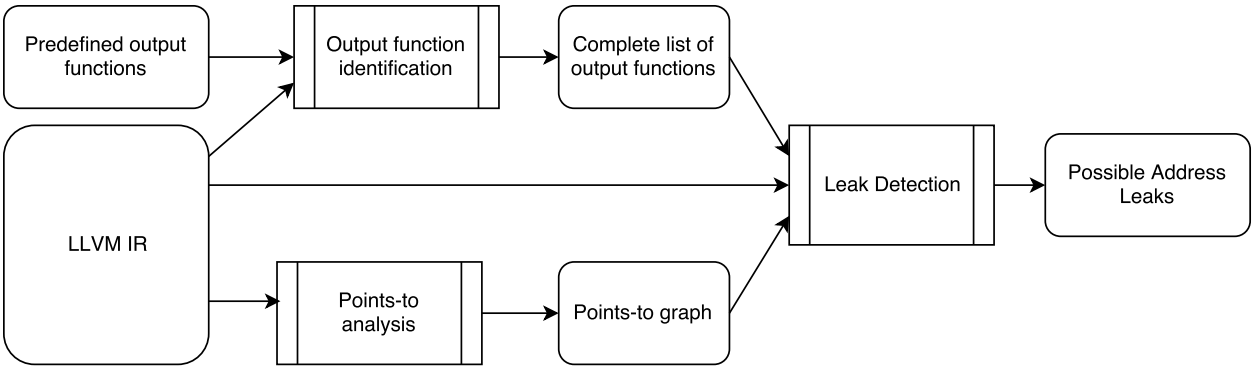

Figure 5.1: Sequence of operations in KALD

5.3 Design

We designed KALD as a tool that statically analyzes the kernel source code to find code that may leak the KASLR offset. Since KALD performs static analysis, it can achieve complete program coverage and detect leaks even on rarely executed code paths. This is not possible with alternative methods like fuzzing or taint analysis, which can only find issues on code that is actually executed. Figure 5.1 illustrates the different steps of this static analysis. KALD begins by finding and marking all calls to output functions. These are functions that can write data to user-visible locations, such as the register context or user-space memory.

Next, KALD runs a points-to analysis to calculate the set of memory locations each program value may point to. KALD then examines all calls to output functions and queries the results of the points-to analysis to determine if any of the function arguments passed at the call site are sensitive values. If a call to an output function may leak sensitive values, then we mark the call site as potentially dangerous.

5.3.1 Output Function Identification

While the kernel can directly write data to any location, it is customary and highly advisable to use the standardized interface when accessing user-space memory. This interface ensures

that the currently running process can legally access a location, and that no memory errors occur (e.g., because the target page is swapped out to disk). The interface includes API functions such as `copy_to_user`, which can copy data to any virtual memory page mapped into the user-space program, as well as functions to write to the `proc` file system, functions to write to the user-space register context, etc.

We compiled a list of these standardized functions, and use this list to seed our output function analysis. For each function in the list, we keep track of its name, location, as well as all the necessary data needed by later stages of our analysis such as which arguments the function copies to user memory, and whether the arguments are copied as a value or dereferenced as a pointer. In this list, we also indicate if the function may perform string formatting, and, if so, which fixed argument contains the format string.

We then analyze the kernel source code to find additional calls to output functions. We are specifically interested in calls to format string functions such as `printk`. Some of the format string functions write data into the system log file, which can be accessed from user-space. These functions can leak the KASLR offset, but they only leak pointers printed out using certain format specifiers (e.g., the integer format specifier `%ld`, or the pointer format specifier `%p`).

Many of the other format specifiers are harmless, as they cannot reveal the literal value of a pointer argument. The `%s` specifier, for example, indicates that the pointer argument points to a string, and that the string will be printed but the pointer itself will not be. Linux's core format string functions also support the `%pK` specifier, which does print out pointers, but obfuscates their value if the pointer points to kernel-space memory, and the currently running user-space program does not have sufficient privileges[1].

Our output function identification step attempts to parse format strings to identify pointer

[1]Specifically, in order to see the real pointer values, the process must have the `CAP_SYSLOG` capability, and it must not have changed its `uid` since it originally started.

```
void my_printk(struct device* dev,
               const char* fmt, ...)
{
  struct va_format vaf;
  va_list args;
  va_start(args, fmt);
  vaf.fmt = fmt;
  vaf.va = &args;

  if (dev) {
    printk("%s: %pV", dev_name(dev), &vaf);
  } else {
    printk("(no dev): %pV", &vaf);
  }

  va_end(args);
}
```

Listing 5.1: A minimal example using the the `%pV` format specifier.

arguments that could leak to user-accessible locations. This is usually possible because most format strings are constants, and can thus be parsed at compile time. For non-constant format strings, we conservatively assume that all of the pointer arguments to the format string function can potentially leak to user-space.

One particularly challenging aspect of distinguishing harmless format string function calls from potentially dangerous calls is the Linux kernel-specific `%pV` format specifier, which is used to perform recursive string formatting. The corresponding argument of this format specifier is a pointer to a `va_format` structure. This struct contains two fields: a pointer to a format string, and an argument list. The format string function recursively substitutes the `%pV` specifier by the format string in its corresponding `va_format` structure, and it also substitutes the corresponding argument itself by the argument list specified in the `va_format` structure. Listing 5.1 shows a minimal code example that uses the `%pV` format specifier.

Our analysis inspects every call to a format string function, parses the format string, and searches for the `%pV` format specifier. When that format specifier is found, `KALD` identifies the

caller of the function, and determines the source of the format string and argument list. We then add the call to the format string function to our list, along with the expanded format string and the expanded function argument lists.

The output function identification step of KALD is Linux-specific. KALD could be ported to analyze other operating system kernels by identifying the relevant API functions within those operating systems, and handling any platform-specific behavior of these functions.

5.3.2 Points-to Analysis

KALD performs a full-program points-to analysis. Points-to analysis is a static program analysis that computes an approximation of the set of objects that a pointer can refer to [93]. We use a field-sensitive version of Andersen's algorithm [5] because there is a high-quality implementation available for LLVM [98]. Our tool can theoretically work with any points-to analysis, as long as the analysis can provide the set of objects referred to by each pointer. KALD could, in other words, easily be adapted to use a different algorithm, allowing it to be improved if a more sophisticated algorithm is available.

5.3.3 Leak Detection

After the points-to analysis completes, KALD inspects each function call site to determine if it could leak any sensitive addresses. If the call site targets a function in the list of output functions, KALD checks each of the arguments to determine if any may leak a sensitive address. If an argument may leak a sensitive address, the call site is flagged as a potential leak location. The list of output functions indicates whether the output argument is used directly as a value, as in the case of the first argument of `put_user`, or dereferenced as a pointer, for example the second argument of `copy_to_user`. KALD uses this information to

properly analyze the call site.

If the argument is used as a value, KALD will indicate a potential leak if it is a pointer containing a sensitive address. If the argument is dereferenced, KALD will examine the pointed-to objects and will indicate a potential leak if any of these objects may be a sensitive pointer, or if any are a struct or buffer which may contain a sensitive pointer.

As we described in Section 5.3.1, calls to format string functions require special treatment as the format string itself determines whether the function may output pointer values. KALD parses this format string during the output function identification step (see Section 5.3.1), and identifies which arguments may leak sensitive address values.

Using the results of the points-to analysis and the list of functions that can copy data to the user, KALD can check if calls to output functions may leak addresses to user programs. However, analyzing a large code base such as the Linux kernel using a sound may-alias analysis can result in false positives if an overly-conservative points-to result says that a pointer may reference a large number of objects. In order to limit false positive rate, we apply a type-based heuristic to the points-to results. We use the actual pointer type available in the code, and compare it with the set of objects that the points-to results indicate may be referenced by the pointer. If the types do not match, then we remove that element from the results. To avoid overly restricting the points-to set, we do not require that the types are exact matches. For scalar types we allow any type that can be losslessly cast to the target type. For struct types we use the common initial sequence criteria. As defined by Yong et al., two structures have a common initial sequence if, for a sequence of initial fields, all corresponding fields have compatible types [109]. If the type of the actual argument is a common initial sequence of the type of the referenced object, then we consider the types to match. For array types we apply the relevant criteria to the array element type, including if the element type is itself an array type, thereby handling nested array types.

Since we use a field-sensitive analysis, we are also able to apply this heuristic to pointers that may refer to sub-fields of global structures. The points-to analysis provides the index into the referenced object. If the type of the field at that index is compatible with the type of the actual argument, then we consider the types to be compatible.

The type-based heuristic allows us to successfully limit the results of the points-to analysis, but there are cases where the type information is either not available, or is not useful. This is the result of generic pointers (`void*`) and character pointers (`char*`), which may point to objects of any type, even under the strict aliasing rule in C. For generic pointers, we conservatively assume that any type is a possible target, and do not limit the results based on type.

Using a type-based heuristic can result in false negatives, particularly for a program that is not type-safe. As a result, `KALD` will not detect an address leak that is the result of type confusion. We do not assume that the Linux kernel is completely type safe, but we consider this a worthwhile trade-off to reduce the number of false positives. Other kernel static analysis tools have also found this trade-off to be desirable [76]. Finding type confusion bugs is outside the scope of this work.

5.4 Implementation

We implemented `KALD` as an analysis pass that operates on LLVM Intermediate Representation (IR) code [68]. We use Clang, the C/C++ front end of the LLVM project, to compile C source code into LLVM IR, and `llvm-link`, the LLVM linker, to link multiple LLVM IR files into a single IR file.

`KALD` utilizes pointer analysis to detect pointers to kernel code and data. We used the implementation of Andersen's analysis [5] from the open source SVF framework [98] as the

underlying pointer analysis of our implementation.

The input to `KALD` is the LLVM IR of a program. `KALD` operates as a module pass, analyzing the entire LLVM module it is given as input. We analyze each call site that appears in the LLVM IR to determine if it could potentially leak a kernel address based on the function being called and the arguments at that call site. However, there are also some kernel interfaces for writing data to user-space that may be implemented with inline assembly code. For example, on both x86 and ARM64, `put_user` is implemented as a C macro that inserts inline assembly code. `KALD` handles these cases by recognizing the assembly code for these operations. Within LLVM IR, inline assembly appears as a call site targeting an `InlineAsm` value. The `InlineAsm` object contains the string of the assembly code that will be inserted. `KALD` will check for inline assembly that could leak information by matching the assembly code with a set of known assembly code strings.

`KALD` can process IR files of any size, but performing global analysis on large code bases can be time and memory intensive. To improve the usability of `KALD` for frequent analysis during the development cycle, `KALD` has the option to limit the analysis time. However, limiting the analysis time can result in false negatives (missed issues) if the analysis does not converge within the allowed time. Alternatively, users can reduce the analysis time by limiting the size of the input by running `KALD` incrementally on submodules of a program individually. Smaller programs will be more likely to converge within the allowed time, however, this may also introduce false negatives since the complete program was never analyzed as a whole. `KALD` does not try to model any code that is not available for analysis, so when analyzing submodules individually, `KALD` can find issues resulting from behavior within the code being analyzed, but will not detect leaks that only result from cross-module interactions.

Limiting the analysis time or analyzing submodules of a program is a source of false negatives, but not of false positives. A false negative may occur when a pointer could contain a sensitive address, but the analysis is unable to process the relevant assignments to determine the full

points-to set, either due to exceeding the time budget or not having all the necessary code made available for analysis. However, this cannot cause false positives because limiting the analysis can only result in smaller points-to sets that may not contain all of the possible targets for a pointer.

5.5 Results

We evaluated `KALD` by running it on the source code of Linux 4.14, with patches applied to make it fully compatible with Clang [64]. To maximize the coverage of our analysis, we built the kernel with the maximal set of configuration options enabled (allyesconfig) for x86_64. We used Whole Program LLVM[2] to automatically collect the LLVM IR files created by Clang and link them together. In our evaluation we analyzed the kernel code on a per-subdirectory basis. For each top-level kernel subdirectory other than drivers, we analyzed all of the code within that directory in one pass. For the drivers subdirectory, due to the large amount of code, we analyzed each subdirectory within drivers individually. We analyzed a total of 143 modules and `KALD` identified 408 potential address leaks. These results are summarized in Table 5.2. To estimate the number of true issues, we randomly selected 40 of the reported issues and manually verified them. Of these, we found 8 cases where a sensitive address could leak, and 32 cases that were false positives. We measured the running time and peak memory use of the analysis of each of the 143 modules, and report the maximum, median, and average of the 143 uses of `KALD`.

The primary sources of false positives are cases where the analysis is overly conservative and determines that the pointer can refer to a large number of objects, and there is no type information available to reduce the number of targets. In these cases it is difficult to determine the actual referents of the pointer.

[2]https://github.com/travitch/whole-program-llvm

Table 5.2: Summary of Results

Total reported issues	408
Estimated true issues	81
Total analysis CPU time	23h 43m
Maximum analysis wall clock time	15h 19m
Median analysis wall clock time	1.8 s
Average analysis wall clock time	9m 57s
Maximum memory use	97.8 GB
Median memory use	305 MB
Average memory use	3.0 GB

Another source of false positives is that the pointer analysis does not take control flow into account. One of the false positives was a case where a struct is copied to user memory using `copy_to_user` as a result of an `ioctl` system call. During the processing of the system call, a kernel address is written to one of the fields of the struct, but before it is copied to the user, that field is cleared. The analysis we used is not flow-sensitive, so it cannot determine that this is not a true leak. However, this was not a significant source of false positives, so while `KALD` could benefit from a flow-sensitive analysis, the algorithm we used still provided useful results.

5.5.1 Kernel Address Leaks Detected by `KALD`

In this section we examine two real address leaks that were found using `KALD`. These are previously unreported leaks[3], and are presented to highlight how different aspects of our design help to discover leaks.

The first is a leak of an address through a call to `printk`. Listing 5.2 shows a simplified version of the vulnerable code. Within function `uart_remove_one_port`, `KALD` indicates that the fourth argument to `dev_alert` may leak the global variable `cdns_uart_port`. The function `dev_alert` is an output function that is not in our initial list of known output

[3]We have reported the exploitable leaks discovered by `KALD` to the Linux kernel developers.

```
static struct uart_port cdns_uart_port[2];

int
uart_remove_one_port(struct uart_driver *drv,
                     struct uart_port *uport) {
  struct uart_state *state;
  struct uart_port *uart_port;
  state = drv->state + uport->line;
  uart_port = uart_port_check(state);
  if (uart_port != uport)
    dev_alert(uport->dev,
              "Removing wrong port: %p != %p\n",
              state->uart_port,
              uport);
}

void
dev_alert(const struct device *dev,
          const char *fmt, ...) {
  struct va_format vaf;
  va_list args;
  va_start(args, fmt);
  vaf.fmt = fmt;
  vaf.va = &args;
  if (dev) {
    /* elided */
  } else
    printk("%s(NULL device *): %pV", level, vaf);
  va_end(args);
}

static int
cdns_uart_remove(struct platform_device *pdev) {
  struct uart_port *port = pdev->dev.driver_data;
  rc = uart_remove_one_port(&cdns_uart_uart_driver,
                            port);
  return rc;
}

static int
cdns_uart_probe(struct platform_device *pdev) {
  struct uart_port *port;
  int id;
  id = of_alias_get_id(pdev->dev.of_node, "serial");
  port = &cdns_uart_port[id];
  pdev->dev.driver_data = port;
  return 0;
}
```

Listing 5.2: Simplified source code resulting in a leak of the KASLR offset through `printk`.

functions, but was detected automatically in the output function identification step. Within `dev_alert`, `printk` is called using the `%pV` format specifier. The `uport` pointer comes from the caller of `uart_remove_one_port`. This function is called by `cdns_uart_remove`, which gets the port from a field within a structure. This field is set within `cdns_uart_probe`, and is set to be a pointer to an element within `cdns_uart_port`. Therefore, when `uart_remove_one_port` is called from `cdns_uart_remove`, the address of a global variable can be leaked.

This example shows the combined benefits of our output function identification step and the interprocedural points-to analysis. The output function that leads to the leak is not in the predefined list of output functions, but `KALD` automatically adds it in the first step of its analysis. The interaction that leads to the leak is the result of interprocedural behavior of multiple functions. Detecting this leak requires interprocedural analysis.

The other leak occurs due to a call to `copy_to_user`, as shown in Listing 5.3. This vulnerability was reported to kernel developers and has been acknowledged and assigned CVE-2018-7755 [33]. Function `fd_ioctl` implements the `ioctl` interface for floppy disk drives. This driver contains the `FDGETPRM` ioctl, which copies a `floppy_struct` to user-space address `param`. The `floppy_struct` exists in the global array `floppy_type`, so the address of the struct itself is a sensitive address. However, when used as an argument to `copy_to_user`, the pointer is not output directly. Instead, the memory it refers to is copied. When `KALD` analyzes a call to `copy_to_user`, it checks if the object referenced by the second argument contains any pointers. In this case it does contain a sensitive pointer. The final field of a `floppy_struct` is a pointer, a `char*`, which points to a string representing the name of the device. The elements of the array `floppy_type` are initialized with pointers to constant strings within the kernel's memory. When the `copy_to_user` executes, this kernel address will be leaked to user-space memory.

`KALD` can detect this leak since it models the behavior of different output functions and distinguishes between pointers output directly and pointers that are dereferenced. This, in concert

```
static struct floppy_struct floppy_type[32] = {
  {    0,  0,0,  0,0,0x00,0x00,0x00,0x00,NULL    },
  {  720,  9,2,40,0,0x2A,0x02,0xDF,0x50,"d360" },
  /* elided */
};

static int
fd_ioctl(struct block_device *bdev, fmode_t mode,
         unsigned int cmd, unsigned long param) {
  int type = ITYPE(UDRS->fd_device);
  int size = sizeof(struct floppy_struct);
  const void *outparam;

  switch (cmd) {
  case FDGETPRM:
    outparam = &floppy_type[type];
    break;

  default:
    return -EINVAL;
  }

  copy_to_user((void *)param, outparam, size);
  return 0;
}
```

Listing 5.3: Simplified source code of a leak of the KASLR offset through `copy_to_user`.

with the points-to analysis, allows KALD to determine that in this case the copy_to_user call will leak a kernel address.

We empirically verified this leak by sending the FDGETPRM ioctl to the floppy driver on a virtualized system with a virtual floppy drive. The returned struct does contain a kernel pointer. Across multiple boot cycles on a system with KASLR enabled, the relative offset between the returned pointer and kernel code pointers remained unchanged. This demonstrated that the returned pointer was randomized with the KASLR offset and could be used to bypass KASLR.

5.5.2 Finding Known Leaks

We also verified that KALD can detect previously reported address leaks. We identified two recently reported direct pointer disclosures, obtained the kernel versions containing the vulnerabilities, and analyzed the vulnerable code with KALD.

The first was a leak of static variables through a call to seq_printf within the pm_qos module [47]. The function seq_printf is a string formatting function that can be used to format output to proc files. This leak is the result of a programmer mistakenly using the format string %pk with a lower case k for a kernel pointer instead of the correct %pK. While this could be confusing for a human reviewer, KALD correctly parses the format string and determines from the points-to analysis that the argument is a kernel pointer. Since the analysis is able to determine that this call could leak a kernel pointer, KALD reports a potential leak.

The other leak we examined was a printk call within the function acpi_smbus_hc_add. In this case, the leaked address was the address of a struct acpi_ec. These structs are allocated from the kernel heap in the function acpi_ec_alloc. Since this call site can only

leak a heap address, KALD does not report a potential issue. Heap addresses are outside the scope of what KALD is designed to detect because the heap is not randomized with the same KASLR offset as the code. Finding a heap address will not provide an attacker with the information they would need to construct a code reuse exploit. However, the pointer analysis used by KALD tracks heap objects, so not detecting heap address leaks is not a limitation of our approach, but rather an intentional design choice in order to focus on address leaks that are most likely to be useful in an attack.

5.6 Limitations

KALD requires whole program interprocedural points-to analysis, which is both time and memory intensive, especially for programs as large as the Linux kernel. There have been recent proposals to allow interprocedural program analyses to scale to programs the size of the Linux kernel. These include using a disk-based graph solving system to overcome memory limitations [103] or partitioning the kernel along system call boundaries in order to decompose the analysis to elements of a manageable size [45]. KALD could utilize these techniques to lower the time and memory requirements of the pointer analysis phase of operation.

A conservative points-to analysis can also lead to a high false positive rate, especially for code bases as large and complex as Linux. A more precise pointer analysis will reduce the false positive rate, but that could further increase the time and memory required for analysis.

The issue of false positives can also be mitigated by improving the usability of the tool and making it easier for users to identify true issues. This can be done by creating heuristics to sort, filter, and prioritize results to show the issues most likely to be true. The tool could also be extended in order to make it easier for developers to verify reported issues by providing

a more detailed report indicating which program statements resulted in the points-to result that is reported.

5.7 Conclusion

Pointer disclosure vulnerabilities can be exploited by adversaries to bypass KASLR. While there have been efforts to address sources of pointer disclosures, these techniques do not address detect direct disclosure vulnerabilities where the kernel outputs code pointers on purpose.

We presented the Kernel Address Leak Detector (`KALD`), a tool that statically analyzes the Linux kernel source code to detect direct pointer disclosure vulnerabilities. `KALD` compiles a list of functions that can leak information to user-space programs, and uses the results of a points-to analysis to determine whether specific invocations of these functions can disclose kernel code pointers based on the arguments passed to the function. `KALD` detected several direct disclosure vulnerabilities in the Linux kernel.

Chapter 6

Related Work

6.1 Control Data Attack Mitigations

Control data protections are effective at stopping code reuse attacks, although they have limited applicability to non-control data attacks. They can broadly be classified into enforcement based and randomization based protections. Enforcement mechanisms enforce a security policy, while randomization based approaches randomize low level details of a program in order to provide a probabilistic protection against attacks.

An example of an enforcement based mitigation is control flow integrity (CFI), first proposed by Abadi et al. in 2005 [1]. Control flow integrity enforces that the control flow taken at run-time is valid. This is enforced by inserting a check at every indirect control flow transfer to ensure the target is an allowed location. There are two types of indirect control flow transfers the "backward" edges which are return instructions and the "forward" edges which are indirect calls and jumps. These are two separate problems, the forward edges must be present on the static control flow graph of the program, but the backward edges must be paired with the most recently called function. Integrity of the forward edges can be enforced

by inserting code to check if the target is a member of the allowed set of targets. However, the same mechanism is overly permissive for backward edges. Instead, they can be verified using a shadow stack. The shadow stack is a stack in a protected memory region separate from the program's run-time stack. Valid return addresses are stored to the shadow stack, and before a return is executed, the return address on the normal stack is compared to the value on the shadow stack. Protecting forward edges by checking the target and backward edges using a shadow stack stops most forms of code reuse attacks including ROP.

The original CFI work sparked a great deal of research interest in the design space of CFI mechanisms. The efforts to improve CFI include proposals to adapt CFI to protect C++ virtual dispatch [111, 50, 100], apply CFI using binary rewriting [101, 102], enforce CFI using cryptography [77], and integrate CFI with fine-grained randomization [79]. CFI has also found wider adoption outside of the research community. Two of the most popular open source compilers, gcc and LLVM, now contain CFI implementations [100].

The CFI mechanism available in gcc is a policy known as virtual-table verification (VTV), a form of C++ aware CFI. A CFI implementation benefits from being aware of the semantics of C++ because this information allows it to correctly determine the valid targets of virtual functions calls. A C++ aware CFI system can stop attacks that construct fake VTables to redirect control flow. Before a virtual call, VTV verifies that the VTable pointer in the object agrees with the type of the object. VTV has an advantage over most other CFI implementations in that it places no restrictions on incremental compilation or dynamic code loading. Both incremental compilation and dynamic loading pose challenges for CFI enforcement because the complete code is never available for analysis. VTV includes enough information to reconstruct the type hierarchy at run-time, and uses this information to verify the targets of virtual calls.

LLVM contains a CFI implementation known as indirect function-call checks (IFCC). Unlike VTV, IFCC does not depend on the details of any high level language, but instead constructs

jump tables for all indirect call targets. All function pointers are modified to refer to the jump tables, and all indirect calls are modified to ensure that they go through the jump tables. This substantially reduces the number of valid call targets and limits attackers to functions available in jump tables with the correct type.

Another related policy is object type integrity (OTI) [14]. OTI differs from CFI in that the former protects objects identities while the latter verifies the targets of indirect control flow. It is designed to stop attacks that tamper with virtual table pointers. Virtual table pointers are a field in a C++ object which points to that object's vtable, which in turn contains function pointers to virtual functions. Since vtable pointers must be in writable memory, attackers can overwrite the vtable pointer of a valid object, or construct a fake object with a virtual table pointer of their choosing. An attacker may then set the virtual table pointer to a different vtable in the program, or a vtable the attacker has injected. OTI is able to stop these attacks by enforcing that the vtable pointer used for dynamic dispatch matches the one written to the object in the constructor function. This policy will not only stop attacks that perform vtable pointer overwrites, but also those that use injected objects, since an injected object will not be created using a constructor. OTI is also more precise than even C++ aware CFI because, statically, the allowed set of targets at a virtual call site may be large, but at run-time, there is one valid target, namely the function corresponding to the run-time type of the object. To enforce OTI, all constructor functions are modified to write the vtable pointer to a secure metadata region, and then during dynamic dispatch, the program uses the vtable pointer from the metadata region.

Another set of policies that can protect against code reuse attacks are code-pointer integrity (CPI) and code-pointer separation (CPS) [65]. These related policies ensure the integrity of code pointers in the program. CPI is the more strict policy, it ensures that all code pointers and any pointer that can be used to reach a code pointer cannot be modified by an attacker. CPS relaxes this policy somewhat to reduce run-time overheads, only ensuring

that code pointers cannot be directly modified, but pointers that may refer to code pointers are unprotected. The enforcement mechanism for CPI and CPS is the same, they place all protected pointers in a safe metadata region. The metadata region contains the pointer value along with bounds information for that pointer. All reads or writes of protected pointers access them from the metadata region and these accesses are bounds checked to protect against overflow. Additionally, to protect return addresses, the stack is separated into a safe stack and an unsafe stack, and any object that could overflow is placed on the unsafe stack. CPI provides complete protection against code reuse attacks because an attacker will be unable to modify any control data to divert control flow.

An alternative to enforcement-based defenses are randomization based defenses. Randomization based strategies randomly change details of the program to make low-level details such as gadget locations unpredictable. The most widely deployed randomization based defense is address space layout randomization (ASLR) [85]. ASLR has been implemented in all major operating systems for both user space applications and the operating system kernel. With ASLR enabled, when a program is loaded, the base addresses of different memory regions are randomly chosen, including the location of executable code. This random offset will shift gadgets to unpredictable locations, stopping code reuse attacks. However, the randomization provided by ASLR is relatively coarse grained, if an attacker can discover a single pointer to a known memory region, they can derandomize that entire memory region.

There are also more fine-grained randomization techniques that randomize more than the base address. One such technique is random insertion of NOP instructions throughout the program's code. This can be done with low run time overhead by using the results of profiling information to guide the concentration of added instructions [54]. Another option is to randomize the location of every instruction within the range of the entire address space [52]. Fine-grained code randomization can also be applied to code generated by just-in-time compilers [53]. Fine-grained randomization introduces substantially more entropy to

the code layout than ASLR, so it is no longer enough to discover a single pointer, instead an attacker will need to disclose much more information to discover the location of gadgets needed for an attack.

However, an attacker who is able to disclose the code can still mount a code reuse attack [94]. To address this weakness, there have been efforts to construct randomization defenses that remain effective in the face of information disclosure. Isomeron is one randomization strategy that is resilient to code disclosure [37]. Isomeron maintains two copies of the program, one that is randomized with fine-grained ASLR, and one that is not. At each control flow instruction, the implementation randomly switches between the different copies of the program. An attacker that knows the entire code layout will still not be able to determine which version of the program is selected after each control flow instruction.

Another approach to combat information disclosures is enforcing execute only memory for code. There are usually no restrictions on disclosing code because while memory used to hold executable code is not writable, it is generally readable. If the readable permission is removed from code than an attacker cannot leak it. One system that uses execute only code to provide protection against code disclosure is Readactor [26]. Readactor stops code disclosure by completely separating code and data and enforcing execute only permissions for all memory used to store code. It also provides fine-grained code randomization, and further hides the code layout by using trampolines placed in execute only memory instead of function pointers. An attacker will have no knowledge of the code layout, and no way to discover any information about the code. There have also been efforts to enforce execute-only memory on systems without memory management units [13] and to enable execute-only code for legacy binaries [21].

6.2 Non-control Data Attack Mitigations

All of the defenses we have discussed so far primarily protect against control data attacks such as ROP. Non-control data attacks do not overwrite control data or cause invalid control flow. Policies like CFI, OTI, or CPI do not stop these attacks. Code randomization can mitigate some non-control data attacks if it also changes the location of data, for example ASLR will randomize the base addresses of the heap, stack, and global variables. However, this is still vulnerable to derandomization through information disclosure, and because data memory must be both readable and writable, it is generally not possible to stop data disclosure. Additional defensive techniques have been developed to protect programs against non-control data attacks. These defenses are generally comprehensive data protections, so they protect both control data and non-control data, so they are also effective at mitigating code reuse attacks in addition to non-control data attacks.

One enforcement based defense is data-flow integrity (DFI) [17]. In many ways DFI is analogous to CFI, but applied to data flows instead of control flows. DFI determines the valid data-flow graph of a program and then instruments the program to ensure that data-flows at run-time are valid. The data-flow graph is constructed by using static points-to analysis to compute reaching definitions and construct the data-flow graph. The enforcement mechanism utilizes a secure metadata region that associates metadata with every memory location. Store instructions are instrumented to update the metadata for the target location with a definition identifier. Load instructions are then instrumented to check the definition identifier stored in the metadata region and raise an exception if it is not a member of the set of valid definitions for that location. This is also applied to control data, including return addresses, so DFI effectively stops non-control data and code reuse attacks that violate the DFI policy. An important consideration is that the policy is computed using static program analysis, and they use a conservative analysis. This means that DFI may miss some attacks that do not violate the statically determined policy.

Another enforcement based approach is write integrity testing (WIT) [3]. WIT enforces the policy that the target of a memory write is in the set of allowed objects. The policy is again based on static analysis. First they use points-to analysis to determine the set of objects that each write instruction can target. Then they assign a color to each object and write instruction such all targets of a write instruction are assigned the same color. At run-time they track the color of memory locations using a metadata region known as the color table, which maps a one byte color identifier to each eight-byte slot of memory. The color table is updated as objects are allocated, and writes are checked against the color table. WIT will stop all attacks that violate the write integrity policy, but the number of attacks stopped depends on the precision of the analysis. They use a context-insensitive points-to analysis to determine the write integrity policy, and can only support a maximum of 256 colors.

There have also been randomization based defenses, for example data space randomization (DSR). DSR was independently and concurrently proposed by Cadar et al. [15] and Bhatkar and Sekar [11]. The objects in the program are classified into equivalence classes based on the result of static analysis, each equivalence class is assigned a different encryption key, and all memory accesses are encrypted by XORing with that key. These DSR systems both used a context insensitive analysis to construct equivalence classes, and exclude equivalence classes that cannot be the base of overflows. Like DFI and WIT, the number of attacks that are stopped depends on the precision of the static analysis. The classification of equivalence classes defines a policy of allowed data flows, and any attack which relies on a forbidden data flow will be stopped. However, since this is a randomization based defense, the attack is not detected by the system, rather the results of any illegal overwrite will be unpredictable, and the attacker will not be able to gain control the system.

DSR randomizes the representation of data in memory, but it is also possible to randomize data in other ways. Lin et al. propose data structure layout randomization (DSLR) as a method to add fine-grained randomness to the data layout of a program [72]. They randomize

the layout of data structures and stack frames by reordering fields and inserting padding fields. This is selectively applied to a subset of data structures, if a data structure is used as part of a library or network interface, it will not be randomized because the receiver of the data would not know the layout. The randomization is applied during compilation, and then the layout remains fixed. DSLR is not a comprehensive protection, it is only intended to provide protection against attacks that rely on the layout of data structures.

Chen et al. extend the idea of DSLR with the notion of adaptive data structure layout randomization [19]. Instead of randomizing data structure layout during compilation, data structures are randomized at run-time. Their system modifies the program during compilation to enable run-time layout randomization. They modify all data structures to self-randomize and modify all data structure accesses to get the field through the randomized offset and to periodically re-randomize the data structure. This is possible because each data structure has associated metadata describing the current layout of the data structure. The metadata also allows the system to derandomize a data structure if needed, for example if it is passed as an argument to a library function. Since the system is able to continuously rerandomize, it is resilient to an attacker that is able to interact with a system and leak the data structure layout. It also improves on static DSLR because the defense can be applied to all data structures in a program, and data structures will only be derandomized when necessary.

6.3 Memory Safety Enforcement

An alternative to exploit mitigation is full memory safety enforcement. Memory safety makes memory corruption impossible, so it prohibits any attack that utilizes a memory corruption vulnerability. There are two primary components of memory safety, spacial safety and temporal safety. Spacial safety ensures that no data structure will be accessed outside

the valid bounds of the data structure, while temporal safety ensures that no data structure will be accessed outside of its valid lifetime. Violations of either spacial or temporal safety can be used to mount attacks. Memory safety enforcement provides a deterministic protection and usually does not rely on the precision of static analysis. A number of memory safety mechanisms have been proposed. However, some of these mechanisms cannot handle memory reallocation correctly [38]. Others are incompatible with unprotected external code [106].

Softbound [81] is a compile-time transformation for spacial safety enforcement that uses a disjoint metadata store to maintain compatibility with unprotected external binaries. The metadata is per-pointer, indexed by the memory location where the pointer is stored, and contains base and bounds information for each pointer. All memory accesses are bounds checked using the base and bounds associated with the pointer operand. Loads of pointer values also load the base and bounds from the metadata region, and stores of pointer values update the metadata for the address where the pointer is stored. Softbound maintains metadata separately from pointers so the memory layout is preserved. Even in the presence of arbitrary casts, Softbound provides complete spacial safety because a cast can never corrupt metadata. However softbound only enforces spacial safety, it does not provide enforcement of temporal safety.

There are also proposals to use low-fat pointers to track pointer metadata [41, 40]. Low fat pointers encode the base and bounds directly in the value of the pointer, so there is not a separate metadata region.

CETS is a compiler-based system to enforce temporal safety [82]. It provides complete detection of all temporal safety violations including dangling pointer accesses, double `free`'s, and invalid `free`'s. CETS implements its checks using a lock-and-key mechanism. It uses a disjoint metadata region to store per-pointer metadata containing a key and a lock location. If the pointer is valid, the data stored in the lock location will match the key. When a pointer is `free`'d, the lock is invalidated. CETS enforces temporal safety by checking the

value of the lock and key before each pointer dereference. Using disjoint metadata maintains compatibility because pointer representation is unchanged.

In the absence of spacial safety errors, CETS is formally proven to enforce temporal safety. To realize this level of assurance, CETS can be combined with a spacial safety enforcement system such as Softbound. Softbound+CETS provides complete spacial and temporal memory safety, preventing all memory corruption, and therefore stopping any memory corruption exploits. However, this level of security comes at a high run time performance overhead, Nagarakatte et al. report 116% overhead for a subset of the SPEC2006 benchmarks.

6.4 Hardware Assisted Approaches

There has been research to develop hardware assisted approaches for different protection mechanisms. Adding additional capabilities the processor can accelerate common sources of overhead or allow for implementation of protection techniques that would be difficult with software alone. For example, WatchdogLite extends the x86_64 ISA with additional hardware to efficiently enforce both spacial and temporal memory safety [80]. They provide additional instructions to accelerate the common operations of pointer based checking with disjoint metadata. Specifically they provide three new types of instructions that allow for efficient metadata loading and storing, spatial checking, and temporal checking. Notably, their ISA extension does not add any additional hardware structures, all new instructions use existing architectural registers. WatchdogLite provides efficient protection with a minimal hardware extension by splitting responsibility between the compiler and the hardware. The hardware only provides instructions that allow for fast metadata management and checks, the compiler is responsible for associating metadata with pointers and inserting instructions to perform checks and manage metadata. The compiler is also able to optimize the program to eliminate redundant or unnecessary checks. This hybrid solution allows them to substantially reduce

the overhead of memory safety enforcement. The average overhead is 29% for full spacial and temporal memory safety enforcement.

Hardware features for memory protection are now even present in commercially available processors. In 2013 Intel introduced memory protection extensions (MPX) [59], an extension to the hardware architecture to support pointer bounds checking. Like WatchdogLite, it is designed to implement pointer-based bounds checking with disjoint metadata. However, unlike WatchdogLite, MPX does not support checking for temporal errors. MPX adds new registers to hold pointer bounds and new instructions to perform bounds checks and manage metadata. An important feature of MPX is its backwards compatibility. All of the new MPX instructions decode as NOPs on processors that don't support MPX, providing complete backwards compatibility.

In 2017 Oleksii et al. released a technical report thoroughly evaluating MPX and comparing it to alternative bounds checking implementations [84]. They found that while MPX is a promising technology, it still has many limitations. MPX still has a substantial run time performance overhead of approximately 50% on average. MPX is also unable to provide protection against temporal errors. They also find that MPX does not support multithreading, when applied to a multithreaded program, MPX may encounter both false positives and false negatives. Although it is not yet a completely robust solution, by choosing to create MPX, Intel has shown that hardware modifications to improve security can make commercial sense.

There have also been proposals for hardware assisted CFI enforcement. It is possible to use existing hardware features to enforce CFI, for example GRIFFIN and FlowGuard [44, 73] use Intel Processor Trace to trace control flow and detect CFI violations, and PathArmor [102] uses the Last Branch Record registers for a similar purpose. Another approach is to extend the hardware with specialized features to support CFI. HAFIX [35, 36] is an ISA extension designed to protect both forward and backward edges on embedded platforms. HAFIX uses a state model with per-function labels to enforce CFI. It ensures that calls can only

target valid function entry points, and returns must target call preceded instructions in a currently executing function. HAFIX adds special landing pad instructions for function calls and returns, and call and return instructions must now target landing pads. When a call instruction is executed, it causes a processor mode switch such that the only valid instruction is the call landing pad. The call landing pad instruction puts the processor back into normal mode, and activates a label for the currently executing function. There is also another new instruction that deactivates the current function's label. This is inserted before return instructions. Then when a return is executed, the processor again performs a mode switch requiring a return landing pad and the return landing pad checks that the label of the targeted function is active. This is a coarse-grained policy, calls can target any valid function entry, and returns can return to any actively executing function, not just the most recent caller. Return instructions can also target any call site within any of the actively executing functions, not just the most recently executed call site. Although the policy is coarse-grained, it still substantially reduces the number of possible targets for any indirect control flow transfer, and provides CFI enforcement with about 2% run time overhead.

Hardware support for CFI has also transitioned from research prototypes to features available in commercial processors. In 2016 Intel announced Control-Flow Enforcement Technology (CET) [60], an ISA extension to provide support for forward and backward edge CFI. Forward edges are handled using a state model with special landing pad instructions similar to HAFIX, and backward edges are protected by using a hardware-managed shadow stack. Like MPX, Intel has maintained backwards compatibility by selecting instructions for CET that are NOPs on machines that do not support CET. While hardware-based solutions have higher barriers to deployment, MPX and CET show that vendors are willing to implement these features because of the possibility for increased security with low overhead.

There are also hardware assisted data protections. Hardware-assisted data-flow isolation (HDFI) [95] is a system for efficiently implementing data isolation, which can then be used

to implement other security policies. It accelerates data-flow integrity style checks. Write operations update a tag for the memory location being written, and read operations check that the tag is valid. HDFI only supports one bit tags, so there are only two possible tag values. This is not enough to implement fine-grained DFI enforcement, but it is sufficient to implement many common security policies, and can be efficiently supported with a small hardware modification. Each memory word is associated with a one bit tag, and the hardware automatically manages storing and updating the tags. The authors of HDFI demonstrate its utility by implementing several security policies including a shadow stack and CPS.

Yang and Shin propose using a hypervisor to encrypt memory pages to provide memory secrecy from the operating system and other processes [108]. Similar to HARD, this technique uses hardware (hypervisor mode) to support data encryption. However, an attempt to extend their technique to provide intra-process data isolation would change the page lifetime assumptions of their paper substantially, and incur substantial performance and memory overhead.

Works such as SeCage [74] or Intel's MPK [61] are designed to restrict memory access to protect secrets. These techniques could be used to control access to the encryption keys in HARD. However, these systems are primarily intended for infrequently used secrets, while HARD doesn't consider any data "secret" and encrypts all program data.

6.5 Exploitation Techniques

All of the work on defensive techniques was motivated by advances in exploitation techniques, which in turn has motivated new types of exploits. One important class of attacks are code reuse attacks. In code reuse attacks, an attacker diverts control flow to instruction sequences already present in the program to perform malicious actions. An example is return-

oriented programming (ROP) [91, 87]. A ROP attack ruses small code sequences usually ending in return instructions, although it is also possible to use other indirect control flow instructions [18]. These sequences are called *gadgets*, and by linking gadgets together as a sequence of return addresses on the stack, the attacker is able to construct an exploit. ROP is an incredibly powerful attack primitive, many applications contain a Turing complete set of gadgets. Researchers have developed tools that will automatically find these gadgets and use them to construct exploits [89, 87, 42, 55]. While ROP is a powerful and sophisticated attack, it leads to control flow that is significantly different from valid executions so CFI enforcement will stop ROP attacks. A ROP attack also depends on knowing the exact address of gadgets, so code randomization is also an effective countermeasure.

To overcome randomization based defenses, Snow et al. propose the notion of just-in-time code reuse and JIT-ROP attacks [94]. This attack technique bypasses fine-grained code randomization. Snow et al. demonstrate that if an attacker can start from a single known-valid code address, they can disassemble the entire memory page containing that address, then harvest additional code pointers from the disassembled code and perform recursive disassembly to discover a large portion of the code without the risk of causing memory errors. The attacker can then search this code for gadgets and construct an exploit. However this depends on reading the code, so a system that enforces execute-only memory like Readactor [26] can stop JIT-ROP.

Another advanced code reuse attack is counterfeit object-oriented programming (COOP) [88]. A COOP attack abuses C++ dynamic dispatch on attacker created objects. The attack makes use of a loop in the program that traverses a collection of C++ objects and performs a virtual function call on each. The attacker supplies a collection of specially crafted objects to this loop, and by choosing the layout of the objects, and the values of the virtual table pointers, the attacker can perform arbitrary computation and invoke arbitrary system calls. A COOP attack doesn't exhibit any of the identifying characteristics of other types of code reuse at-

tacks, and in fact has control flow and data flow that is similar to benign C++ execution. Consequently, COOP attacks are not stopped by coarse-grained CFI techniques that are not aware of C++ semantics. However, fine-grained, C++ aware CFI enforcement like VTV [100] and virtual table pointer protections like OTI [14] can stop COOP attacks.

It is also possible to mount attacks that do corrupt control flow. These attacks are termed non-control data attacks because they do not overwrite control data, instead they overwrite other security-critical data. We have known that non-control data attacks could pose a serious threat for quite some time. In 2005 Chen et al. demonstrated that non-control data attacks can be just as severe as code reuse or other control-data attacks [20], leading to arbitrary code execution in the worst case. They demonstrated several attacks that did not alter control flow, but still exploited the program by overwriting security critical data. These attacks all required manual analysis of the program, in contrast to a technique like ROP where there are tools to automatically find gadgets and compile exploits from a high level language [89, 87, 42, 55]. However, non-control data attacks are effective even with CFI or fine-grained code randomization.

More recent research has developed ways to largely automate constructing non-control data attacks. Data-flow stitching [56] automatically searches the data-flow graph for locations where an edge can be added to create a new data flow needed for an attack. This ultimately allows an attacker to automatically construct a new data-flow path, creating data-flow from a vulnerability to some target data. Data-flow stitching creates a systematic way of constructing non-control data exploits, making it easier for attacker to develop such exploits.

Data-oriented programming (DOP) [57] builds upon data-flow stitching to create a non-control data attack roughly analogous to ROP. A DOP attack uses data-oriented gadgets, which are small code sequences that perform primitive operations on data. Many common programs contain a set of data-oriented gadgets that are powerful enough to perform Turing complete computations. However, an attacker cannot force the program to execute data-

oriented gadgets by diverting control flow, instead, an attacker uses a gadget dispatcher loop. This is a loop structure in the program that processes attacker-controlled data, and has some selector that allows the attacker to execute different DOP gadgets on different iterations of the loop. DOP creates a principled technique for constructing powerful non-control data attacks, and these attacks will completely bypass any CFI implementation.

Another attack that is able to bypass CFI is control flow bending (CFB) [16]. CFB is a hybrid between a control data and non-control data attack. Under CFB, an attacker can overwrite control data to alter the control flow, but only to control flow that obeys CFI. How restrictive this is depends on the CFI enforcement technique. They evaluate attacks under a very restrictive policy called fully-precise static CFI. Under this policy, a control flow transfer is only allowed if it is found in some non-malicious execution of the program. This is the most precise policy that does not alter a program's behavior, and is more precise than any existing CFI implementation. Even with fully-precise CFI, it is possible to mount a CFB attack. However, fully-precise static CFI only considers edges present in the static control flow graph, it does not require that a function must return to its caller. CFB attacks are more difficult if the system uses a shadow stack to check return addresses, but they are still possible.

6.6 Static Bug-finding

Exploit mitigations and memory safety enforcement stop exploits at run-time, but it is also possible to find errors earlier in the program's life cycle, using static analysis. A static analysis tool analyzes the program's source code to identify potential errors, allowing developers to fix these issues before they ever have the possibility of being exploited. This has an advantage over any run-time protection in that it introduces zero overhead. However, it is not possible to detect all errors using static analysis, many techniques have high false positive

rates, and some analysis algorithms are unable to scale to large programs.

One static analysis system that seeks to address scalability problems is Graspan [103]. Graspan is a disk-based graph system designed for scalable program analysis. It is motivated by the fact that many program analyses can be formulated as graph reachability problems. Graspan is a generic system that presents a programming model which takes an input graph from any source and a grammar specifying the rules for an analysis. The system is then able to efficiently apply these rules to find a solution. Graspan achieves scalability on large graphs by using a disk based system to operate on graphs larger than can fit in memory. Graspan is not constrained to any single analysis algorithm, instead the grammar definition determines what analysis is computed. The authors construct a graph generator for LLVM IR and grammars for points-to and data flow analysis and show Graspan is able to compute these analyses on large programs such as the Linux kernel. They also implement several checkers that use the analysis results to find common programming issues including memory errors such as null pointer references or use-after-free.

Another proposal to address scalability concerns specifically for the Linux kernel is K-Miner [45]. K-Miner constructs a graph of the assignment statements and then automatically partitions this graph along the boundaries of system calls. Each system call is then analyzed separately. This reduces the size of any individual graph to be analyzed, and also models the way user space applications interact with the kernel. After partitioning the graph, they compute points-to and data flow analysis, and then implement checkers for common memory errors. While the system is able to analyze the kernel along the system call interface, the false positive rate of their checkers is quite high.

DR. CHECKER is another system designed to statically find bugs in the Linux kernel [76]. They specifically target kernel drivers since these are often a source of vulnerabilities. To simplify the analysis, they do not perform a sound analysis, instead they make assumptions to reduce the necessary number of iterations which reduces the running time and false positive

rate. Under these assumptions they perform field- and context-sensitive points-to and taint analysis, and use these results for a variety of vulnerability checkers. They implement several checkers that utilize these results, and are able to use DR. CHECKER to find previously unknown security critical bugs in Linux kernel drivers.

KINT uses static analysis to find integer bugs in C programs [104]. KINT successfully identified several exploitable bugs in the Linux kernel. Similar to `KALD`, KINT uses heuristics to prioritize and filter its analysis results.

APISAN builds a database of (likely) correct API usage patterns by symbolically executing code and inferring semantic relationships between API calls [110]. APISAN can then statically analyze additional code and cross-check its API usage patterns with the database.

UniSan seeks to preserve the integrity of the KASLR offset [75]. It focuses on preventing leaks through uninitialized values, while `KALD` finds errors where developers mistakenly output kernel addresses. UniSan has both an analysis component that identifies possible issues, and a run-time protection component that mitigates them.

Chapter 7

Conclusion

Programs written in unsafe languages like C and C++ are prone to memory corruption errors. These memory errors can be exploited by attackers, and sophisticated attack techniques like return-oriented programming and data-oriented programming allow an attacker to perform arbitrary computations and assume complete control of a vulnerable program. In the face of such an adversary we need effective ways to secure programs written in unsafe languages. We explored how static program analysis can be applied to provide protection against memory corruption issues.

Exploit mitigations are one technique that has been proven effective at raising the bar for attackers, and static analysis can be used to determine a policy that is used by a run-time mitigation. However common code-centric mitigations fail to address non-control data attacks. These attacks can be just as damaging, but do not introduce invalid control flow, and therefore are able to bypass common mitigations. Data space randomization is one approach that is able to stop non-control attacks, and relies on static analysis to construct equivalence classes of variables within a program. DSR will only stop an attack if the data an attacker overwrites is in a different equivalence class than legitimate targets of the vulnerable

instruction. Therefore, if the equivalence classes are more precise, more attacks will be stopped. We presented context-sensitive DSR, a version of DSR that provides stronger protection than all prior DSR implementations. Context-sensitive DSR uses a context-sensitive points-to analysis to construct equivalence classes, which allows context-sensitive DSR to construct a greater number of more fine grained equivalence classes. As a result context-sensitive DSR will stop more attacks than other DSR techniques. We also encrypt all possible equivalence classes to protect against all types of memory errors and always use 8-byte encryption keys.

The increased protection of context-sensitive DSR does lead to higher run-time performance overheads. To address performance concerns we developed HARD, which adapts context-sensitive DSR to target specialized hardware instructions that accelerate the operations needed to implement DSR. This allows HARD to provide precise protection with low overhead. The specialized hardware also frees the software implementation from having to manage encryption keys and protects the encryption keys from leakage and tampering.

We also applied static analysis to address a common weakness of ASLR, namely its vulnerability to pointer disclosures. A pointer disclosure allows an attacker to determine the layout of the code and bypass ASLR. We developed `KALD`, a static analysis tool to help developers discover direct pointer disclosures in operating system kernels. `KALD` analyzes the kernel source code and uses the results of points-to analysis to determine locations that may leak sensitive pointers. Using `KALD` we detected several direct disclosure vulnerabilities in the Linux kernel.

In this dissertation we presented applications of points-to analysis to address security problems. We improved the precision, security, and performance of data space randomization, and developed the first tool to detect direct disclosure vulnerabilities in operating system kernels. It is our hope that these contributions will lead to a more secure software landscape.

Bibliography

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):4:1–4:40, Nov. 2009.

[3] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with wit. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 263–277, May 2008.

[4] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 7(49), Aug. 1996.

[5] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Cophenhagen, 1994.

[6] S. Andersen and V. Abella. Data execution prevention. changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies, 2004.

[7] ARM architecture reference manual ARMv8, for ARMv8-A architecture profile, 2015.

[8] D. Avots, M. Dalton, V. B. Livshits, and M. S. Lam. Improving software security with a c pointer analysis. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 332–341, May 2005.

[9] G. Balatsouras and Y. Smaragdakis. Structure-sensitive points-to analysis for c and c++. In *International Static Analysis Symposium*, pages 84–104. Springer, 2016.

[10] B. Belleville, H. Moon, J. Shin, D. Hwang, J. M. Nash, S. Jung, Y. Na, S. Volckaert, P. Larsen, Y. Paek, and M. Franz. Hardware assisted randomization of data. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, 2018.

[11] S. Bhatkar and R. Sekar. Data space randomization. In D. Zamboni, editor, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 5137 of *Lecture Notes in Computer Science*, pages 1–22. Springer Berlin Heidelberg, 2008.

[12] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 30–40, New York, NY, USA, 2011. ACM.

[13] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi. Leakage-resilient layout randomization for mobile devices. In *NDSS*, 2016.

[14] N. Burow, D. McKee, S. A. Carr, and M. Payer. Cfixx: Object type integrity for c++ virtual dispatch. In *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[15] C. Cadar, P. Akritidis, M. Costa, J.-P. Martin, and M. Castro. Data randomization. Technical Report MSR-TR-2008-120, Microsoft Research, September 2008.

[16] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, Washington, D.C., 2015. USENIX Association.

[17] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 147–160, Berkeley, CA, USA, 2006. USENIX Association.

[18] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572. ACM, 2010.

[19] P. Chen, J. Xu, Z. Lin, D. Xu, B. Mao, and P. Liu. A practical approach for adaptive data structure layout randomization. In G. Pernul, P. Ryan, and E. Weippl, editors, *Computer Security ESORICS 2015*, volume 9326 of *Lecture Notes in Computer Science*, pages 69–89. Springer International Publishing, 2015.

[20] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 12–12, Berkeley, CA, USA, 2005. USENIX Association.

[21] Y. Chen, D. Zhang, R. Wang, R. Qiao, A. M. Azab, L. Lu, H. Vijayakumar, and W. Shen. Norax: Enabling execute-only memory for cots binaries on aarch64. In *2017 IEEE Symposium on Security and Privacy (SP)*, 2017.

[22] S. Christey and R. A. Martin. Vulnerability type distributions in CVE. `http://cwe.mitre.org/documents/vuln-trends/index.html`, May 2007.

[23] Codenomicon and N. Mehta. The Heartbleed Bug. `http://heartbleed.com/`, 2014.

[24] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 952–963, New York, NY, USA, 2015. ACM.

[25] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Usenix Security*, volume 98, pages 63–78, 1998.

[26] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A. R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *2015 IEEE Symposium on Security and Privacy*, pages 763–780, May 2015.

[27] CVE-2015-0089. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0089`, 2015.

[28] CVE-2015-1097. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1097`, 2015.

[29] CVE-2015-8569. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8569`, 2015.

[30] CVE-2017-1000410. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1000410`, 2017.

[31] CVE-2017-14954. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-14954`, 2017.

[32] CVE-2018-5750. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-5750`, 2018.

[33] CVE-2018-7755. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-7755`, 2018.

[34] M. Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 35–46, New York, NY, USA, 2000. ACM.

[35] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin. Hafix: Hardware-assisted flow integrity extension. In *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, pages 74:1–74:6, New York, NY, USA, 2015. ACM.

[36] L. Davi, P. Koeberl, and A.-R. Sadeghi. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 133:1–133:6, New York, NY, USA, 2014. ACM.

[37] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming.

[38] D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for c with very low overhead. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 162–171, New York, NY, USA, 2006. ACM.

[39] W. Dietz. Building poolalloc with current llvm development branch? `http://lists.llvm.org/pipermail/llvm-dev/2015-May/086213.html`, 2015.

[40] G. J. Duck, R. H. Yap, and L. Cavallaro. Stack bounds protection with low fat pointers. In *Symposium on Network and Distributed System Security (NDSS)*, 2017.

[41] G. J. Duck and R. H. C. Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 132–142, New York, NY, USA, 2016. ACM.

[42] T. Dullien, T. Kornau, and R.-P. Weinmann. A framework for automated architecture-independent gadget search. In *Proceedings of the 4th USENIX Workshop on Offensive Technologies (WOOT)*, 2010.

[43] evasi0n iOS 7.0.x jailbreak - official website of the evad3rs. `https://bugs.chromium.org/p/project-zero/issues/detail?id=967`, 2013.

[44] X. Ge, W. Cui, and T. Jaeger. Griffin: Guarding control flows using intel processor trace. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 585–598, New York, NY, USA, 2017. ACM.

[45] D. Gens, S. Schmitt, L. Davi, and A.-R. Sadeghi. K-miner: Uncovering memory corruption in linux. In *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[46] Google Project Zero. Multiple overflows in tsp sysfs "cmd store". `https://bugs.chromium.org/p/project-zero/issues/detail?id=967`, 2016.

[47] Google Project Zero. Samsung: Kaslr bypass in "pm_qos". `https://bugs.chromium.org/p/project-zero/issues/detail?id=971`, 2016.

[48] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard. Kaslr is dead: long live kaslr. In *International Symposium on Engineering Secure Software and Systems (ESSoS)*, 2017.

[49] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 368–379, New York, NY, USA, 2016. ACM.

[50] I. Haller, E. Göktaş, E. Athanasopoulos, G. Portokalidis, and H. Bos. Shrinkwrap: Vtable protection without loose ends. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, pages 341–350, New York, NY, USA, 2015. ACM.

[51] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using cla: A million lines of c code in a second. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 254–263, New York, NY, USA, 2001. ACM.

[52] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. Ilr: Where'd my gadgets go? In *2012 IEEE Symposium on Security and Privacy*, pages 571–585, May 2012.

[53] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. Librando: transparent code randomization for just-in-time compilers. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, CCS '13, pages 993–1004, New York, NY, USA, 2013. ACM.

[54] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided automated software diversity. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–11, Feb 2013.

[55] A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz. Microgadgets: Size does matter in turing-complete return-oriented programming. In *WOOT*, pages 64–76, 2012.

[56] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. Automatic generation of data-oriented exploits. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 177–192, Washington, D.C., Aug. 2015. USENIX Association.

[57] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *IEEE Symposium on Security and Privacy*, 2016.

[58] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space aslr. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[59] Intel Corporation. Introduction to intel memory protection extensions. `https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions`, 2013.

[60] Intel Corporation. Control-flow enforcement technology preview. `https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf`, 2017.

[61] Intel Inc. Intel 64 and IA-32 Architectures Software Developers Manual, 2017.

[62] Y. Jang, S. Lee, and T. Kim. Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 380–392, New York, NY, USA, 2016. ACM.

[63] Y. Jang, S. Lee, and T. Kim. Drk: Breaking kernel address space layout randomization with intel tsx. *BlackHat USA*, 2016.

[64] M. Kaehlcke. Clang patch stacks for lts kernels and status update. `https://lkml.org/lkml/2017/11/22/943`, Nov 2017.

[65] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 147–163, Broomfield, CO, Oct. 2014. USENIX Association.

[66] W. Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, Dec. 1992.

[67] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. Sok: Automated software diversity. In *2014 IEEE Symposium on Security and Privacy*, pages 276–291, May 2014.

[68] C. Lattner and V. 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*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[69] C. Lattner and V. Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 129–142, New York, NY, USA, 2005. ACM.

[70] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 278–289, New York, NY, USA, 2007. ACM.

[71] D. Lea. A memory allocator, 1996.

[72] Z. Lin, R. Riley, and D. Xu. Polymorphing software by randomizing data structure layout. In U. Flegel and D. Bruschi, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 5587 of *Lecture Notes in Computer Science*, pages 107–126. Springer Berlin Heidelberg, 2009.

[73] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan. Transparent and efficient cfi enforcement with intel processor trace. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 529–540, Feb 2017.

[74] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1607–1619, New York, NY, USA, 2015. ACM.

[75] K. Lu, C. Song, T. Kim, and W. Lee. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 920–932, New York, NY, USA, 2016. ACM.

[76] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna. DR. CHECKER: A soundy analysis for linux kernel drivers. In *26th USENIX Security*

Symposium (USENIX Security 17), pages 1007–1024, Vancouver, BC, 2017. USENIX Association.

[77] A. J. Mashtizadeh, A. Bittau, D. Mazieres, and D. Boneh. Cryptographically enforced control flow integrity. *arXiv preprint arXiv:1408.1451*, 2014.

[78] M. Matz, J. Hubicka, A. Jaeger, and M. Mitchell. System v application binary interface, 2013.

[79] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz. Opaque control-flow integrity. In *NDSS*, volume 26, pages 27–30, 2015.

[80] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Watchdoglite: Hardware-accelerated compiler-based pointer checking. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 175:175–175:184, New York, NY, USA, 2014. ACM.

[81] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 245–258, New York, NY, USA, 2009. ACM.

[82] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Cets: Compiler enforced temporal safety for c. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 31–40, New York, NY, USA, 2010. ACM.

[83] Nergal. The advanced return-into-lib (c) exploits: Pax case study. *Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e*, 2001.

[84] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer. Intel MPX explained: An empirical study of intel MPX and software-based bounds checking approaches. *CoRR*, abs/1702.00719, 2017.

[85] PaX Team. PaX aslr. `http://pax.grsecurity.net/docs/aslr.txt`, 2003.

[86] T. Rains, M. Miller, and D. Weston. Exploitation trends: From potential risk to actual risk. In *RSA Conference*, 2015.

[87] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, Mar. 2012.

[88] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 745–762, May 2015.

[89] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *USENIX Security Symposium*, 2011.

[90] F. J. Serna. The info leak era on software exploitation. *Black Hat USA*, 2012.

[91] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[92] D. Shen. Defeating samsung knox with zero privilege. *BlackHat USA*, 2017.

[93] Y. Smaragdakis and G. Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2(1):1–69, 2015.

[94] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. *2012 IEEE Symposium on Security and Privacy*, 0:574–588, 2013.

[95] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek. Hdfi: Hardware-assisted data-flow isolation. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 1–17, May 2016.

[96] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[97] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *European Workshop on System Security (EuroSEC)*, 2009.

[98] Y. Sui and J. Xue. Svf: Interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 265–266, New York, NY, USA, 2016. ACM.

[99] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, May 2013.

[100] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955, San Diego, CA, Aug. 2014. USENIX Association.

[101] V. v. d. Veen, E. Gktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 934–953, May 2016.

[102] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical context-sensitive cfi. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 927–940, New York, NY, USA, 2015. ACM.

[103] K. Wang, A. Hussain, Z. Zuo, G. Xu, and A. Amiri Sani. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 389–404, New York, NY, USA, 2017. ACM.

[104] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Improving integer security for systems with kint. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 163–177, Hollywood, CA, 2012. USENIX.

[105] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović. The RISC-V instruction set manual, volume I: User-level ISA, version 2.0. Technical Report UCB/EECS-2014-54, UCB, 2014.

[106] W. Xu, D. C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of c programs. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, pages 117–126, New York, NY, USA, 2004. ACM.

[107] W. Xu and Y. Fu. Own your android! yet another universal root. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2015.

[108] J. Yang and K. G. Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, pages 71–80, New York, NY, USA, 2008. ACM.

[109] S. H. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 91–103, New York, NY, USA, 1999. ACM.

[110] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik. Apisan: Sanitizing API usages through semantic cross-checking. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 363–378, Austin, TX, 2016. USENIX Association.

[111] C. Zhang, S. A. Carr, T. Li, Y. Ding, C. Song, M. Payer, and D. Song. Vtrust: Regaining trust on virtual calls. In *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[112] X. Zheng and R. Rugina. Demand-driven alias analysis for c. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 197–208, New York, NY, USA, 2008. ACM.