Michael Rodler, BSc

# Enforcing Pointer Integrity through Static Analysis

**MASTER'S THESIS**

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

**Graz University of Technology**

Supervisor

Prof. Stefan Mangard

Mario Werner

Institute of Applied Information Processing and Communications (IAIK)

Graz, May 2017

# AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date

Signature

# Abstract

A large part of systems-level and performance critical software is written in C or C++. Bugs triggering memory safety violations still plague many C/C++ code bases and can be used by attackers to manipulate the process by corrupting its memory. Orthogonal to finding bugs, research and industry develop mitigation mechanisms, which minimize the impact of a memory corruption vulnerability. Current mitigation mechanisms mostly focus on preventing control-flow hijacking attacks. However, data-only attacks must also be considered, because they allow an attacker to elevate privileges within the application, break existing mitigation mechanisms, or leak sensitive memory contents.

In this thesis, we evaluate a current mitigation mechanism against data-only attacks, Write-integrity testing (WIT). We implement WIT within the LLVM compiler framework and use our implementation to evaluate the security guarantees of WIT implementations. Our results show that WIT does indeed offer protection against data-only attacks. Our implementation can mitigate all vulnerabilities in 33 of 58 programs out of the cyber grand challenge (CGC) dataset. This thesis presents limitations and caveats for the protection a WIT implementation can offer. Additionally the evaluation revealed various common code patterns that severely reduce the security guarantees of WIT. Furthermore, we show that WIT does not provide comprehensive protection against counterfeit object oriented programming (COOP), a modern control-flow hijacking technique.

**Keywords:** Memory Corruption, Data-Only Attacks, Mitigation Mechanisms, Data-Flow Integrity, Write-Integrity Testing

# Kurzfassung

Ein Großteil der systemnahen und geschwindigkeitskritischen Software wird in C oder C++ entwickelt. Man findet häufig Fehler in C/C++ Programmen, die zu Verletzungen von Memory-Safety führen. Angreifer können solche Fehler ausnutzen um den Speicher des Prozesses zu beschädigen und somit das Verhalten des Prozesses zu manipulieren. Neben dem Auffinden von Fehlern werden Mechanismen erforscht, die den Schaden von Memory-Corruption-Angriffen minimieren. Diese setzen ihren Fokus auf Control-Flow-Hijacking-Angriffe. Data-Only-Angriffe müssen ebenso beachtet werden, da sie einem Angreifer ermöglichen, Privilegien zu erhöhen, existierende Präventationsmechanismen zu umgehen, oder auf sensitiven Speicher zuzugreifen.

Diese Arbeit behandelt einen Präventionsmechanismus gegen Data-Only-Angriffe: write-integrity testing (WIT). Wir implementieren WIT in dem LLVM Compiler-Framework und verwenden unsere Implementierung um die Sicherheitsgarantien von WIT zu evaluieren. Unsere Resultate ergeben, dass WIT tatsächlich Schutz gegen Data-Only-Angriffe bietet. Unsere Implementierung schützt 33 von 58 Programmen aus dem cyber grand challenge (CGC) Datensatz gegen alle bekannten Schwachstellen. In dieser Arbeit werden häufige Muster in Programmcode präsentiert, die dazu führen, dass WIT nur schwache Sicherheitsgarantien geben kann. Darüber hinaus zeigen wir, dass WIT keinen umfangreichen Schutz gegen counterfeit object oriented programming (COOP), eine moderne Control-Flow-Hijacking-Angriffstechnik, bietet.

**Schlagwörter:** Speicherkorruption, Data-Only-Angriffe, Code-Reuse-Angriffe, Präventionsmechanismen, Data-Flow-Integrity, Write-Integrity-Testing

# Contents

# List of Figures

# Listings

# 1 Introduction

We interact almost constantly with some form of Internet enabled device. It has become the norm that we use a PC or notebook at work. Smartphones and tablets allow us to use the Internet when we are on our way. In the era of the Internet of things, even household devices such as TVs, fridges or vacuum cleaner robots are connected to the Internet. Consequently more sensitive data is stored and processed in computer systems. Furthermore, computer systems execute many business-critical processes, often with minimal human monitoring. A flawless IT operation can decide between success and failure of a company. While the use of Internet connected computer systems offers many advantages, it also comes with a significant security risk: computer systems are susceptible to attacks from anywhere in the world over the Internet.

Different events in the last years have shown that many parties are interested in attacking computer systems. Personal computers are regular victims of malware. The SpyEye malware family, for example, stole online banking credentials and credit card details, and caused huge financial losses [Gua16]. The latest trend in malware is called Ransomware [FBI15], which encrypts data on infected computers and only releases the decryption key for a certain amount of money. Users lacking a proper backup are coerced into paying the ransom. Even businesses fall pray to this kind of malware [Her16]. Big companies usually deal with a lot of valuable customer data, which makes them a valuable target for cyber criminals. In 2016, Yahoo admitted the compromise of more than 1 billion user accounts [Thi16]. Similarly Dropbox suffered from a breach in 2012, which resulted in over 68 million email addresses and passwords being leaked [Gib]. Attacks on computer systems have even become part of conflicts between nations. The first prominent example for a presumed nation sponsored malware is Stuxnet and was discovered in 2010 [Sch10]. Stuxnet was engineered to infect PCs used in industrial control systems and then perform sabotage. It is speculated that the intended target for Stuxnet were Iranian nuclear power plants. Since Stuxnet further similarly sophisticated malware has been discovered [Kus13].

Attacks on computer systems are often made possible by programming mistakes, commonly called bugs, in the software that runs on the computer systems. Many big software vendors, such as Microsoft, Google, and Apple, have realized that and offer rewards for researchers that report errors in their software, which can be turned into security vulnerabilities. Bugs in operating systems are handled as especially valuable, because gaining control over the operating system gives an attacker full control of the computer [Pro16]. Vulnerabilities in web browsers are similar valuable, because users are easily tricked into opening malicious websites, where an attack against the browser is started.

Although C/C++ family of programming languages is notorious for its easiness to introduce bugs that can be turned into security vulnerabilities, a lot of software is written in C or C++. This is especially true for the critical components of computer systems, such as web browsers and operating systems. The reason for this choice is the high performance that can be achieved with optimized C/C++ code. Another reason for using C/C++ is compatibility with an existing code base. It is unlikely that major projects will move to memory safe languages, and therefore also the security problems will prevail.

The lack of integrated memory safety makes programs written in C/C++ prone to security vulnerabilities. The C memory model exposes a very low-level view on the memory, that is available to the program. Other programming languages guarantee that a reference to some object never points to something invalid. In the C family of programming languages, objects are usually referenced by pointers, which are the address in the memory of the program. Due to programming mistakes, a pointer can be corrupted and will contain some other memory address. The program will dereference the corrupted pointer and treat the bytes at that address the same way as the bytes at the original address. Errors regarding memory handling can be turned into a class of vulnerabilities categorized as memory corruption [V+12].

In order to turn a bug into a security vulnerability, an attacker must be able to abuse the bug to gain access to information that she is not allowed to know or modify. To exploit a vulnerability the attacker must bring the program into a state that was not anticipated by the developer. In the case of memory corruption exploits, the attacker forces the program into an unintended state by corrupting memory, which the program uses to make a decision. A typical example is to modify the user name or a list of permissions for the current user, which would allow an attacker to elevate her privileges. A common goal of an attacker is to gain arbitrary code execution, meaning she can execute any sequence of instructions in the context of the attacked program. With the capability to execute arbitrary code, any data on the computer system can be accessed in the context of the vulnerable program. Even if there is no data of value on the attacked system, the ability to abuse the overtaken system to launch further attacks is enough motivation for an attack.

Manually searching for memory corruption vulnerabilities in programs is a time consuming task. A well established technique for finding bugs is the use of static source code analyzers. They search for patterns in source code, which can expose problematic behavior. Source code analyzers aid the developer in identifying and correcting mistakes. While source code analyzers can identify problematic patterns, they lack the in depth understanding of the code, which is needed to identify bugs buried deep inside the code. An active research area is developing more advanced techniques to automatically identify bugs in programs. For example, formal verification techniques allow developers to prove that a program has the memory safety property. If memory safety cannot be proven, then a bug is identified as counter example. The effort required to use formal verification to secure a program against memory corruption is very high, making it infeasible for most software projects.

Memory safety violations can be detected during testing. Testing tools such as Valgrind [NS07] and compiler based sanitizers, such as the LLVM Address Sanitizer [Ser+12], detect issues with memory management and addressing during runtime. A successful strategy for finding edge cases that are otherwise poorly tested is fuzz testing [Zal]. A fuzzer randomly mutates a initial set of inputs. It is then observed whether the program under test can process the randomly mutated inputs without crashing. The effectiveness of fuzzing is improved by guiding it by coverage [Zal], by extracting features with static analysis [Raw+17] or even augmenting it with symbolic execution [Ste+16]. The downside of testing is that it is not comprehensive and cannot guarantee the absence of bugs.

Since finding and eliminating all security related bugs is infeasible for larger software projects another approach must be taken in addition. Instead of directly correcting the security relevant bugs, the impact of a bug can be reduced. An attacker is then too constrained to launch an attack with this bug. If the cost of a successful exploit is bigger than the reward the attacker gains, it is unlikely that an attack is started. It is desirable to make it as hard as possible for an attacker to construct a working exploit. The process of increasing resistance against attacks is called hardening and can by applied on different levels in a system. Hardening against memory corruption vulnerabilities is done by introducing exploit mitigation mechanisms. Full memory safety can be retrofitted into C/C++ programs, but comes at a significant overhead during compile and runtime [Nag12]. Therefore an interesting research area is to find mechanisms, which provide a subset of the guarantees of full memory safety at an acceptable performance cost.

Randomization based approaches make memory corruption exploits non-deterministic. The probability of a successful exploit is decreased by introducing randomness in the executed program. Approaches to shuffle around the pieces of code [Kil+06] and data [BS08; XKI03] have been proposed at various levels of granularity. The main advantage of probabilistic defenses is that they usually have low overhead. Because of the probabilistic nature the defenses can be bypassed as soon as the attacker has knowledge of the random values. An attacker can abuse other types of bugs to leak randomized values or simply launch a brute-force guessing attack.

Deterministic defenses do not rely on randomness but introduce mechanisms to prevent a certain attack technique or limit the attackers capabilities. Non-executable memory was introduced to counter attacks that injected executable code into the address space of the attacked program. Subsequently this was bypassed by reusing existing code to perform the desired functionality. Code reuse attacks, such as return-oriented programming (ROP), were shown to give the attacker similar capabilities as code injection in common scenarios. To perform computations ROP attacks execute small snippets of instructions out of their intended order and chain them together using return instructions [Sha07].

Both code injection and code reuse attacks require the attacker to hijack the control flow of the vulnerable program. Control-flow integrity (CFI) is a concept that tries to limit the capabilities of an attacker, even in the case the control of the program has been overtaken [Aba+05]. CFI tries to make sure that the programs instructions can only be executed in their intended order, which prevents most forms of code-reuse attacks. This approach severely limits the attacker. However, it was shown that even perfect CFI leaves an attacker with enough capabilities to construct successful exploits [Car+15].

Other mechanisms prevent an attacker from taking over the control-flow of a program in the first place. To take over the control flow of a program the attacker needs to corrupt a code pointer. Defensive mechanisms, such as read-only relocations or stack canaries, protect different kinds of code pointers. One of the first attack techniques was to abuse missing boundary checks of arrays that are placed on the stack to overwrite the return address [One96]. Subsequently stack canaries were introduced to detect return address overwrites by buffer overflows and abort the program [Cow+99]. Code pointer integrity (CPI) generalized the concept of protecting certain code pointers and introduced memory safety for all code pointers in the program [Kuz+14].

Since the emergence of the protections against control flow hijacking the focus of attack techniques has shifted to data driven attacks. These attacks focus solely on data variables and data pointers. They do not corrupt code pointers and do not violate CFI. Overwriting a code pointer is not the only way to influence the behavior of an attacked program. There is plenty of data which influences the control-flow of a program indirectly, because the program bases decisions on the data. Examples are loop counter variables, data pointers in linked data structures, or user names. Data-oriented programming (DOP) is a data-only attack technique that has been shown to be Turing-complete, which gives an attacker similar capabilities as control-flow hijacking attacks [Hu+16]. Furthermore, data-driven attacks are used to subvert mitigation mechanisms against control-flow hijacks [Eva+15].

Data-flow integrity (DFI) and the follow-up write-integrity testing (WIT) [CCH06; Akr+08] are proposed defense mechanisms that cover both data-oriented attacks and also aim to protect against control-flow hijacking attacks. Both enforce restrictions on pointers in the program, thereby making data-only attacks and code pointer corruption significantly harder. When a pointer is used to load data, DFI checks whether the pointer is really pointing to a memory location that matches one of the possible objects to which the pointer may point to. This set of allowed objects is determined with static analysis during compile time. DFI instruments store instructions to propagate a label for the data that is read. WIT is an optimization of DFI, which performs checks only during memory write operations. This is in a sense similar to CFI solutions, which check the target of call and indirect jump instructions. Unlike CFI solutions, DFI and WIT do not restrict themselves to code pointers, but insert checks for any pointer in the program. This is especially challenging in the face of dynamic memory management used extensively in modern programs.

In summary, we argue that it is crucial for future mitigation mechanisms to consider the data driven approach to attacks. DFI and WIT are expected to offer protection against data-driven attacks, but it is currently unknown how far this protection goes and what problems and limitations implementations of DFI and WIT face.

## Contribution

In this thesis we answer the question, how far the protection offered by WIT goes. We evaluate WIT in regard of security guarantees it offers against modern attacks like counterfeit object oriented programming (COOP) [Sch+15] or DOP [Hu+16]. There is no public implementation of WIT or DFI for user-space programs, therefore we implemented a prototype of WIT within the LLVM [LA04] compiler framework. We use the same type of points-to analysis and a similar shadow memory model as described in [Akr+08]. Subsequently we perform the analysis of the security guarantees based on our prototype. Our results show that while WIT cannot offer the same protection as a solution that introduces full memory safety, it can offer protection against control-flow hijacking and data-driven attacks in many cases with lower performance overhead than full memory safety.

Furthermore, we present a systematic evaluation of our LLVM-based WIT prototype (LLWIT) and a comparison with two state of the art mitigation mechanisms, the CFI solution included in LLVM [Tic+14] and SafeStack [Kuz+14]. We perform the evaluation based on the dataset that was used during the cyber grand challenge (CGC) competition. This dataset consists of programs, which were written for the competition. The programs contain known memory corruption vulnerabilities. The CGC data set offers a wide range of different programs with different kinds of vulnerabilities. Many of the vulnerabilities are modeled after bugs that occurred in real-world software. We test all the programs from the dataset with different mitigation mechanisms enabled and check whether they are still functional and whether their vulnerabilities have been mitigated. We furthermore evaluate how the coloring of WIT are distributed and whether metrics based on the colorings are meaningful as indicators for security guarantees. Our implementation is able to mitigate all vulnerabilities in 33 of 58 usable programs from the CGC dataset. We do not observe a difference in the number of colors for programs with mitigated and unmitigated vulnerabilities.

Using the results of the CGC dataset, we look at cases where WIT fails to mitigate vulnerabilities and cases where CFI offers protection, while WIT does not. Our results show that WIT offers protection against most control-flow hijacking and many data-driven attacks. However we identify several weak spots of WIT and DFI in general. We show that WIT with a field-insensitive points-to analysis does not cover certain attack vectors. More concretely we show that WIT restricts, but does not fully prevent COOP attacks, although WIT was proposed as a countermeasure [Sch+15]. Furthermore, we argue that it is very unlikely that WIT offers adequate protection against DOP attacks on larger programs, because of the way WIT constructs the allowed sets of objects per store instruction.

## Outline

The remainder of this thesis is structured in the following way: In Chapter 2 we give a more distinctive definition of memory safety and an introduction to memory corruption attacks. We introduce attack techniques and the current state of the art in mitigation mechanisms. In Chapter 3 we introduce the concept of DFI and WIT. We describe our prototype implementation in Chapter 4. In Chapter 5 we evaluate and compare the security guarantees of the different mitigation mechanisms and show the shortcomings of WIT. We end with a conclusion in Chapter 6.

# 2 Memory Corruption Attacks and Defenses

The lack of memory safety in the C and C++ languages makes programs written in those languages susceptible to memory corruption bugs. An attacker can abuse memory corruption bugs to modify the state of the running program. The impact of such an attack ranges from a denial of service (DoS) by crashing the program to execution of arbitrary code chosen by the attacker. To achieve the latter an attacker has to carefully craft the state of the program, so that the code of the attacker is correctly executed.

Over the years several exploit mitigation mechanisms have been developed and deployed. Following the deployment of new countermeasures, attack techniques have been created that bypass the mitigation mechanisms. Even though exploits haven not been completely eradicated, the mitigation mechanisms made it significantly harder and subsequently more expensive to create a working exploit.

In this chapter we introduce a taxonomy of safety issues in C/C++ programs and give and overview of attack techniques, that are used to exploit memory corruption vulnerabilities. Furthermore we describe the current state of the art mitigation mechanisms against memory corruption exploits.

## 2.1 Spatial and Temporal Memory Safety

The memory model exposed to the programmer in the C and C++ languages is very low level. The language allows the programmer to handle raw bytes of memory and to interpret the memory as any type of object. The low level access to memory is necessary to implement software components that work with hardware directly or to develop resource efficient software. On the other hand this flexibility introduces the possibility for memory safety violations. Depending on the type of bug one can classify it as a violation of spatial or temporal memory safety. Orthogonal to the problem of memory safety is the weak type safety offered in C/C++.

**Spatial Memory Safety** violations happen when a pointer is dereferenced, that points outside of the bounds of the object associated with the pointer. Spatial memory safety violations can happen when bounds checks are missing. Other reasons for spatial memory safety violations are type casts between incompatible types or dereferencing of invalid pointers, such as uninitialized or NULL pointers.

Listing 2.1 shows an example of a spatial memory safety violation. Arrays in the C language are equivalent to pointers to the first object in the array and the array subscript operator is equivalent to a pointer addition and dereference. The array object has 8 elements, but the loop will index up to the 9th element, resulting in a buffer overflow. In the last iteration of the loop the memory location `array + 8` is accessed, which is beyond the bounds of the `array` object.

```c
char array[8];
for (int i = 0; i <= 8; i++) {  // off-by-one error
  array[i] = '\0';  // last loop will set array[8]
}
```

Listing 2.1: Example of a spatial memory safety violation.

In the C language buffer overflows can occur easily when dealing with strings or byte buffers, due to absence of automatic bounds checking and the poorly designed legacy string handling API of the C standard library [One96].

**Temporal Memory Safety** is the property that no objects are accessed before or after they have been allocated. Dangling pointers to unallocated objects turn into temporal safety violations when the dangling pointer is

```
1 uint32_t *p, *q;
2 char *u;
3 p = malloc(8); // allocate a uint32_t[2] array
4 q = p + 1;  // q references the second uint32_t
5 // ...
6 free(p);
7 u = malloc(8); // likely(p == u)
8 // ...
9 *q = ...  // Use-After-Free Bug: modifies u instead
```

Listing 2.2: Example of a temporal memory safety violation.

dereferenced. This leads to problems when the memory of the freed object has been reallocated.

Listing 2.2 shows an example of a temporal safety violation. First a allocation of 8 bytes happens in line 3. A reference to inside the allocated memory is stored in the pointer `q`. The allocated memory is then freed and subsequently another 8 byte chunk is allocated. Typical `malloc` implementations will reuse the freed block and return it. So when the `q` is dereferenced in the last line, it would actually write to the `u` object.

Memory-safe languages, such as C# or Java, deal with the problem of memory safety by automatically introducing checks at various points in the program or runtime environment. They implicitly associate length fields to array types and automatically performing bounds checks and do not allow uninitialized memory reads. To counter temporal memory safety automatic memory management is used, for example by using garbage collection or reference counting. This way a memory location is only deallocated when no reference exists anymore.

**Type Safety vs Memory Safety** A type system can be used to ensure that only compatible types are used in expression. If a type systems cannot decide whether a type is compatible at compile time, the decision can be postponed to runtime. The C language has no concept of types during runtime. The C++ object model requires that for certain classes the type of

```
struct Foo {
  int a;
}; // sizeof(Foo) == 4
struct Bar : public Foo {
  int b;
}; // sizeof(Bar) == 8
// ...
Foo* f = new Foo();
// ...
Bar* b = static_cast<Bar*>(f);  // invalid downcast
b->a = ... // *(b + 0) ok
b->b = ... // *(b + 4) out-of-bounds
```

Listing 2.3: Weak type safety in C/C++ can cause memory safety issues.

an object can be acquired during runtime. The C-style type casting or the union data structure provide facilities to cast between any types allowing the programmer to break any guarantees by the type system.

For type safety to hold also during runtime, memory safety must be enforced. Otherwise the guarantees of the type system are easily broken by violating memory safety. For example a dangling pointer can be used to overwrite data allocated to another type of object, possibly violating invariants of the type.

The weak type system can also lead to memory safety violations. Listing 2.3 shows an example where the lack of proper type checking results in an out of bounds access. The class `Bar` inherits from `Foo`, so it has the members `a` and `b`. A pointer to a `Foo` is downcasted by the programmer to the `Bar` type, without additional checks. In this case there is actually a `Foo` object, behind the pointer, which does not have the `b` member.

## 2.2 Attack Techniques

To exploit a memory safety violation an attacker has to corrupt or leak internal data. The following section gives an overview of the various attack techniques that have been developed. We discuss the goals of each attack, how the attacks are constructed and what the resulting capabilities of the attacker are. We classify the attacks according to [Sze+13] into attacks that

- Code corruption – attack modifies existing code
- Control-flow hijacking – attack modifies code pointers
- Data-only attacks – data variables are modified
- Information leaks – attack outputs data variables/pointers

To achieve any of the above, the attacker can corrupt as many data pointers as needed.

### 2.2.1 Code Corruption

Once a memory section is executable and writable a memory corruption attack can be used to modify the contents of the section. The integrity of the code is not guaranteed anymore and as soon as the process reaches the corrupted code it is executed. Modern operating systems enforce read-only page permissions on sections containing code, which is a strong mitigation against this attack. On systems that do not have a memory management or protection unit this cannot be adequately enforced.

Programs that include just-in-time (JIT) compilers or bytecode interpreters are also prone to code corruption. JIT compilers are used to speed up the execution of domain-specific languages, as is common in modern web browsers that include a Javascript engine. During code generation the compiler must necessarily map the code section as writable and as long the code section is writable it is also vulnerable to code corruption. In a similar manner bytecode interpreters are prone to code corruption attacks as long as the bytecode is writable. If the bytecode is corrupted, the attacker would gain the same capabilities as a legitimate program running inside the interpreter. In case of general purpose programming languages running inside an interpreter this would mean the same privileges as the interpreter process.

### 2.2.2 Control-Flow Hijacking

Control-flow hijacking summarizes various techniques, that are used by an attacker to take control of the program control flow. All control-flow hijacking attacks are initiated by the corruption of a code pointer. When the corrupted code pointer is used by the program, the control-flow of the program takes a malicious branch. The program then jumps to an arbitrary location in the program, which is chosen by the attacker. By providing the right inputs, the attacker can carefully craft the state of the program so that it does not crash after the malicious control-flow branch. Instead the program continues to execute in an unintended way. The attacker can redirect control-flow to injected code or re-use existing code.

The control-flow graph (CFG) of a program is a directed graph, whose nodes correspond to the basic blocks of the program, and the edges represent the control-flow transfers between the basic blocks. A basic block is a list of consecutive instructions. During execution the program operates within its normal CFG. During a control-flow hijacking attack the attacker corrupts a code pointer and forces the program to perform an unintended control-flow transfer, which is outside of the CFG. Figure 2.1 shows an example of an CFG (blue nodes). During the attack the attacker forces a control-flow transfer from basic block $D$ to $X$, which is not part of the regular CFG. Control-flow hijacking attacks dynamically add edges and sometimes nodes to the CFG.

The first and still prominent technique to hijack the control flow was to overwrite the return address on the call stack. The usual C calling conventions place data and return addresses on the same stack. This makes the return address prone to be overwritten by stack-based buffer overflows, a common error in C programs that manipulate strings stored on the stack, as described in Section 2.1).

Other often targeted code pointers are inside the global offset table (GOT), a section used by the dynamic linker to store pointers to functions with addresses unknown at compile time. During the first call of a library function the dynamic linker is invoked to look up the address of the function and to store it in the GOT. Any call to a function in a dynamically loaded library is using the indirection via the GOT. If an attacker can replace a function

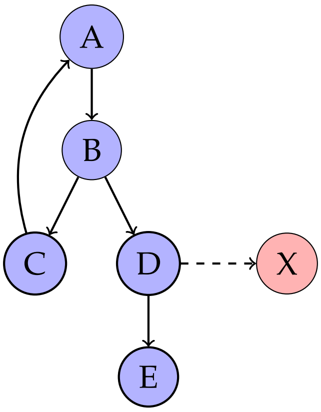


Figure 2.1: Example of control-flow graph. The blue nodes are part of the normal CFG. A unintended control flow transfer from $D$ to $X$ is forced by the attacker.

pointer inside the GOT she can effectively replace a function. For example the attacker replaces the `puts` function with the `system` function. If the attacker controls any of the strings, which is passed to `puts`, she can execute shell commands.

In C++, classes can have virtual methods, which can be overridden by subclasses. On a virtual method call the actual target of the call is chosen based on the actual type of the object during runtime. Since the compiler only knows the static type of the variable, it does not know which implementation of the virtual method must be called. To support this feature virtual calls first perform a lookup in a virtual method table, which is a table of code pointers to the implementations of the virtual methods. Each class has its own virtual method table or `vtable` for short. A pointer to the class' `vtable` is usually placed at the beginning of each object. While the virtual method table itself should be read-only, an attacker can replace the `vtable` pointer to point to a different table of code pointers. Therefore not only code pointers can be abused to hijack control flow, but also pointers pointing to code pointers.

### Code Injection

Early exploits injected small snippets of machine code into the address space of the vulnerable program and redirected control flow there to achieve arbitrary code execution. The code is injected as regular user input and a code pointer corruption is used to redirect control flow to the location of the user input. This small snippets of code usually did nothing more to launch a shell and make it available over the network, so that the attacker can further use the system. The name shellcode has been established for these small snippets of code, because of their typical functionality.

Shellcode is usually very constrained in its size and the possible bytes it might contain. For example shellcode that is processed by C string manipulation functions must not contain zero bytes as this would terminate the string and stop further processing. Further restrictions might be that the shellcode cannot contain a newline character or might even be restricted to alphanumeric characters. All of these restrictions can be bypassed by cleverly writing the shellcode and avoiding certain kinds of instructions [rix01]. It has even been shown that a shellcode can mimic English prose text on `x86_64` [Mas+09].

### Code-Reuse Attacks

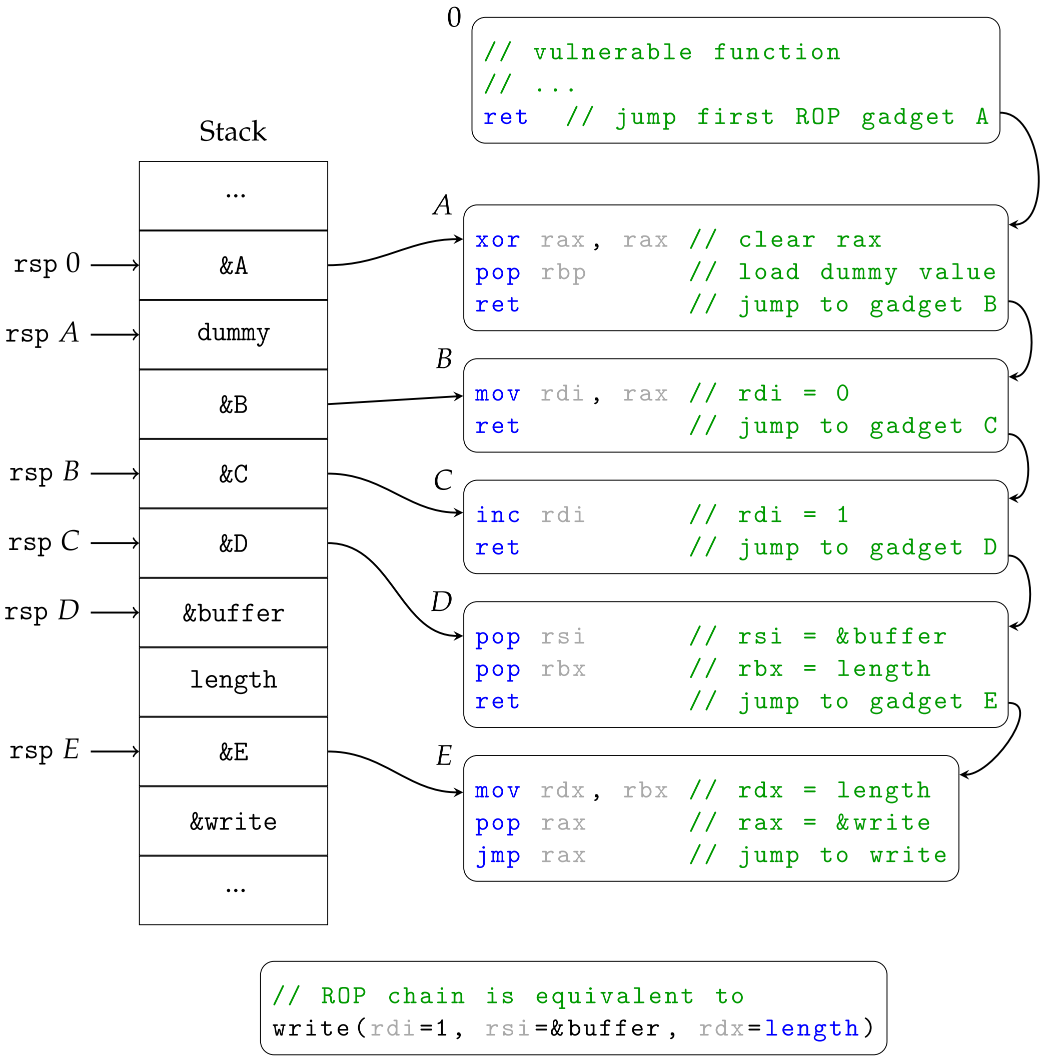
After the injection of shellcode was mitigated by data execution prevention (DEP) or code-signing schemes a class of attack techniques appeared that reuse existing code of the vulnerable program in an unintended way. Shellcode was mostly used to create backdoors into the system by starting a shell process. To setup such a backdoor the shellcode typically binds the shell to another network port or connects back to the attacker. The idea of code-reuse attacks is that the tasks of shellcode can be also performed by code that is already present in the program.

The first code-reuse attack technique was return to lib(c) (ret2libc). The idea of ret2libc attacks is to reuse functions that are always present in the standard libraries. One typical example is the `system` function, which executes the command that is passed as parameter. To perform a ret2libc attack, control over the stack is needed. The return address is overwritten with the address of the function she wants to call. Additionally the stack is prepared in a way that after the return to the target function, attacker controlled arguments to the function are on the stack as expected by the called function. Multiple calls can be chained by ensuring that for the return of the first called function, the stack contains a valid return address and the parameters for the second function. It was later shown that almost any computation can be achieved by chaining together libc function calls [Tra+11].

Return to lib(c) attacks are not so easily possible when calling conventions are used that pass parameters in register and not on the stack. On the 32-bit Intel `x86` architecture the usual calling conventions pass arguments on the stack. The 64 bit extension `x86_64` has switched to a different calling convention, passing arguments in register. Other architectures such as `ARM` and `MIPS` also pass arguments in registers. To still perform ret2libc attacks existing chunks of code must be abused to set the register values [Kra05].

Chaining existing chunks of code has been further generalized to return-oriented programming. Return instruction are used to chain together short pieces of code, called gadgets. Each gadget typically performs a couple of instructions and returns to the next gadget. If enough gadgets exist ROP is Turing-complete. An attacker can use ROP to achieve arbitrary code execution without injecting shellcode into the address space of the vulnerable program [Sha07].

Figure 2.2 shows an example of a ROP attack, where the stack on the left is prepared with gadgets in existing code and data values. The chain of ROP gadgets performs a function call to `write`. To achieve this the attacker corrupts the stack. When the vulnerable function returns in the basic block 0, control passes to the attacker's ROP chain. Since on `x86_64` parameters are passed in register the ROP chain first sets the registers `rdi`, `rsi` and `rsi` to the parameters of the call. In Figure 2.2 the gadgets *A*, *B* and *C* are used to set the register `rdi` for the first parameter to 1. The gadget *D* loads the second parameter, a pointer to a buffer, into the register `rsi`. Gadget *D* also loads the length parameter from the stack, but into the wrong register. The value is moved in gadget *E* into the correct `rdx` register. The last gadget *E* then uses a indirect call to invoke the `write` function.



```
// vulnerable function
// ...
ret  // jump first ROP gadget A
```

```
xor rax, rax // clear rax
pop rbp      // load dummy value
ret          // jump to gadget B
```

```
mov rdi, rax // rdi = 0
ret          // jump to gadget C
```

```
inc rdi      // rdi = 1
ret          // jump to gadget D
```

```
pop rsi      // rsi = &buffer
pop rbx      // rbx = length
ret          // jump to gadget E
```

```
mov rdx, rbx // rdx = length
pop rax      // rax = &write
jmp rax      // jump to write
```

```
// ROP chain is equivalent to
write(rdi=1, rsi=&buffer, rdx=length)
```

Figure 2.2: Example of a short ROP payload, which sets up a call to the `write` function. On the left is the corrupted stack and on the right is the control-flow between the ROP gadgets.

After ROP it was discovered that also other branching instructions can be used to chain together existing code snippets. Using pairs of `pop reg`, `jmp reg` instructions one can emulate the return instruction [Che+10]. Also indirect jumps can be used to achieve something similar to ROP, but is not necessarily constrained to use the stack to dispatch control-flow transfers [Ble+11]. Various variants of ROP have been developed in order to create attack techniques, that are more reliable or resilient against mitigation techniques. Similar computational expressiveness can also be achieved with SROP, which abuses UNIX signal handling and the `sigreturn` syscall to redirect control flow and set registers [BB14]. Attacks like blind ROP [Bit+14] and SROP [BB14] provide means for an attacker to perform attacks when nearly no information is known about the targeted binary. Just-in-time ROP is an attack technique that systematically discovers the executable code using an arbitrary read vulnerability and a leak of one code pointer [Sno+13].

An attack technique specifically targeting C++ code is counterfeit object oriented programming (COOP). This attack does not use the stack to dispatch control flow transfers. Instead the building blocks of COOP are counterfeit C++ objects. Specifically COOP uses virtual methods of C++ classes as gadgets, called `vfgadgets`. Each C++ object that contains virtual method has a pointer to a virtual method table attached. Each call to a virtual method of the object is performed indirectly by loading the code pointer of the respective index from the `vtable`. For COOP attacks, pointers into `vtables` and object members are arranged into, possibly overlapping, counterfeit objects. To direct the control flow to the various gadgets a special dispatcher gadget is used. This dispatcher gadget iterates over the counterfeit objects and calls virtual methods on this objects. Virtual methods that call into two other virtual methods can also be used as recursive dispatcher gadget. In sufficiently large C++ code bases enough gadgets exist to make COOP attacks Turing-complete.

Using COOP has several advantages for an attacker. Heap exploits often have to move the stack pointer into the heap to perform ROP, which can be avoided by COOP. Furthermore it is harder to defend against COOP because mitigation mechanisms need to consider C++ semantics and code pointers are not involved directly, but only indirectly via the vtable [Sch+15; Cra+15a].

### 2.2.3 Data-Only Attacks

For data-only memory corruption attacks, the attacker is only able to change non-control data, i.e. no code pointers. We consider corrupting pointers to code pointer tables, such as vtables, not as an data-only attack. Instead of hijacking the control flow directly via code pointers, the attacker corrupts other parts of the programs data to indirectly influence the control flow. Data-only attacks stay inside the legitimate control-flow of the targeted program.

Every piece of data that influences the control flow of the program is a possible target for such attacks. The feasibility of such attacks was studied in [Che+05]. Examples for possible targets of memory corruption are:

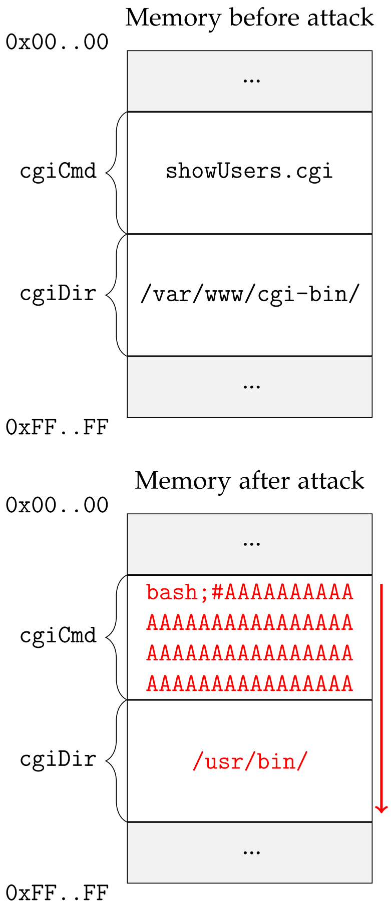
- **Configuration data** contains information on how the program behaves. Corrupting file path information is a prime example on how an attacker can gain access to files, outside the configured file paths. For example a web server only serves files from a certain directory or executes dynamic scripts only in another directory. By corrupting the configured directories the attacker could be able to point the script directory to the user upload directory and achieve command execution.
- **User identity data** is used by the program to decide whether a user is allowed to perform a certain action. By corrupting a username or user identifier an attacker can gain access to resources she would not be allowed to user otherwise.
- **User inputs** are usually validated by the application to ensure that the data matches the expectation. If the attacker can corrupt the data after validation the program might break because certain assumptions about the user input do not hold anymore. This can lead to further memory corruption.

Figure 2.3 shows an example from [Akr+08] for a vulnerability that is exploitable with a data-only memory corruption attack. The short snippet of C code contains a buffer overflow vulnerability, that allows an attacker to corrupt configuration data. The code is part of a web server, that can execute common gateway interface (CGI) scripts. CGI scripts are external programs, that are called by the webserver to dynamically generate content.

Depending on the requested uniform resource locator (URL) a different CGI script is executed by the webserver. The code contains two global variables, called `cgiDir` and `cgiCmd`. Both are arrays of type `char`, i.e. strings in the C language. The `cgiDir` string contains the path to the directory where the CGI scripts are located. The user is allowed to execute any of the CGI scripts. Now we assume that the requested URL is controlled by the attacker and the webserver extracts the last part of the URL to determine which CGI script to call. So the function `ProcessCGIRequest` is called with untrusted input that is controlled by the attacker. Since the `size` variable is also untrusted and can be $> 64$, the snippet is vulnerable to a buffer overflow attack. The attacker can abuse the buffer overflow from the `cgiCmd` variable into the `cgiDir` variable to change the `cgiDir` to a different path. This allows the attacker to execute a program in a different path as a CGI script, leading to a arbitrary command execution. The attack never corrupts any kind of control data and is therefore a data-only attack. This example show the severity of some data-only attacks, as an attacker has gained the same capabilities as with a shellcode injection attack.

Data-only attacks against configuration data or user identity data are only examples for interesting targets. Data-only attacks can be generalized to all data variables, that are used as the condition for a branching instruction. If a data variable is used as a condition for a branch, it can indirectly influence the control-flow of a program. For example Kenali [Son+16] is a defense mechanism against data-only exploits targeting the Linux kernel. First Kenali identifies a set of critical data variables in syscall handlers. It then separates the critical set of variables from the non-critical set using hardware mechanisms. Variables within the critical set are further protected by introducing instrumentation, similar to write-integrity testing (WIT) [Akr+08]. To identify the critical set, they perform static analysis on all syscall handler functions. Every data variable, that is used as condition for a block that returns an error code, is put into the set of critical variables. In general it is hard to automatically decide which parts of a program's data must be considered as security critical.

Although data-only attacks are known since the first publication of memory corruption exploits, research on the expressiveness has only currently begun. Data-oriented programming (DOP) is a data-oriented attack technique, shown to be Turing-complete [Hu+16]. The requirement for a DOP attack to



```c
// written only by ProcessCGIRequest
char cgiCmd[64];
// written only by SetCGIDir
char cgiDir[64];

void ProcessCGIRequest(char* curCmd) {
  // curCmd is untrusted input
  size_t i = 0;
  // buffer overflow here
  while (i < strlen(curCmd)) {
    cgiCmd[i] = curCmd[i];
    i++;
  }
  // system(strcat(cgiDir, cgiCmd))
  ExecuteRequest(cgiDir, cgiCmd);
}

void SetCGIDir(char* newDir) {
  size_t i = 0;
  while (newDir[i] && (i < 64)) {
    cgiDir[i] = newDir[i];
    i++;
  }
}
```

Figure 2.3: C snippet that contains a vulnerability, exploitable with a data-only buffer overflow. The buffer `cgiCmd` can overflow into `cgiDir`, leading to arbitrary command execution.

allow arbitrary computation is that three types of gadgets are available inside of a dispatcher loop: assignment, dereference and addition. The dispatcher loop can be either a loop, whose loop condition can be manipulated to run the whole DOP attack or a repeatedly invoked memory corruption.

Data-only attacks are also used as a first stage in an exploit to set up a control-flow hijack [Eva+15]. To various extent this is already required on modern operating systems. Exploit writers need to bypass various mitigation mechanisms, as discussed in Section 2.3. Vendors of modern web-browsers, like Google Chrome or Microsoft Edge, have invested significant effort into introducing mitigation mechanisms. Current research studies the feasibility of using data-only attacks against hardened web browsers [Gaw+16b; Rog+]

### 2.2.4 Information Leaks

An information leak occurs when the attacker gains access to information, which she is not intended to view. This can happen because of logic bugs, but also due to memory corruption. Information leaks due to memory safety violations are out of bounds reads or reading from dangling pointers. Information leaks can be data-only attacks or can be constructed using a control-flow hijack attack. Furthermore information leaks are frequently part of exploits, as they are needed to bypass probabilistic mitigation mechanisms. [Sha+04; Sno+13].

If a vulnerability can be used to corrupt a pointer, which is then dereferenced and sent back to the attacker, a very powerful information leak is created. This allows the attacker to read almost all contents in the address space of the program. Listing 2.4 shows an example, where a buffer overflow can be used to create a powerful information leak. The pointer `s` can be corrupted with a buffer overflow, triggered by the `gets` function. The `puts` function dereferences the corrupted pointer and prints the memory contents back to the attacker.

An prominent example for an information leak is the notorious heartbleed bug in OpenSSL [Tea]. This bug occurred because of a missing boundary check. The length of a buffer that was sent back to the client, was taken

```c
char input[128];
char *s;
// ...
if (...)
  s = "success";
else
  s = "error";
gets(input);  // triggers buffer overflow
              // which overflows into s
              // s is now controlled by the attacker
puts(s);      // reads string from manipulated s
              // and prints it back to the attacker
```

Listing 2.4: Example of a C program with an information leak of arbitrary addresses

without further checks as the length parameter to a call to `memcpy`. This buffer was then sent back to the client. Although the buffer over-read was limited to 64 kB on the heap, the attack has been shown to be powerful enough to leak private keys and plaintext passwords from production systems [14; Sha14].

## 2.3 Mitigation Mechanisms

Soon after the emergence of the first memory corruption exploits the arms race between attackers and defenders started. On the defensive side, research has focused on how to stop general classes of attacks or make certain types of memory corruption bugs unexploitable. For nearly every countermeasure introduced there were found ways to bypass them. When introducing exploit mitigation techniques it must be ensured that there are no compatibility issues with existing code and furthermore to be adopted the mitigation technique must not introduce too much overhead. In real world systems when choosing between higher security guarantees and lower overhead, the latter is usually chosen. This creates gaps in the security guarantees of a mitigation technique, which can be used by an attacker to bypass it.

Over the years, mitigation mechanisms have made it into major operating systems and compilers. This has made exploitation already significantly harder. Nowadays exploits often abuse more than one bug to bypass the deployed mitigation mechanisms. In the following section, we give an overview over different approaches to mitigating memory corruption exploits. We first discuss one of the most widely deployed mitigation mechanisms: non-executable memory. We describe proposals that introduce full memory safety. Then we discuss probabilistic approaches. We discuss another widely deployed defense, called adress space layout randomization (ASLR), and its shortcomings. Furthermore we elaborate the challenges of more fine-grained code randomization. Then we describe the concept of control-flow integrity (CFI). We end with mitigation mechanisms that protect code pointers to stop control-flow hijacking attacks.

### 2.3.1 Non-Executable Memory

The first buffer overflow exploits also injected shellcode into the vulnerable application [One96]. This shellcode is introduced in the address space of the program via user input, which typically resides in the stack or heap segments. The attacker then redirects control flow into these segments. On processors with virtual memory, permission bits are stored in the pagetable that indicate whether a page is readable or writable. The code segment is usually mapped as read-only, as code changes rarely during execution of a program. To counter code injection, a bit for executable was introduced. This way only the pages explicitly marked as executable can contain code. For CPUs that did not have this feature, techniques to emulate this behavior based on segmentation were developed. Depending on the operating system and CPU vendor this feature is called DEP, No eXecute bit, XD-bit or `W^X`.

After DEP started to become widely deployed the focus shifted to code reuse. Having to resort to code reuse attacks is rarely a serious limitation for an attacker. For example an often targeted function is system in the standard libc, which is used to execute shell commands. While DEP is easily bypassed by code-reuse attacks, it is nevertheless useful as a first line of defense. More advanced mitigation mechanisms often rely on the enforcement of restrictive memory protections.

### 2.3.2 Retrofitting Full Memory Safety

Since the underlying problem of memory corruption exploits is the lack of memory safety, an apparent mitigation is to introduce full memory safety into the C/C++ languages. There are various proposals that try to achieve this. Memory safety approaches are distinguished based on whether they are object-based or pointer-based [Nag12]. Object-based approaches ensure that a pointer only points to its intended referent. Pointer manipulations are checked such that derived pointers stay within the bounds of the original object. Object-based approaches track bounds-information per object. On the other hand pointer-based approaches track base and bound information per pointer. The bound information is updated during pointer arithmetic and checked at pointer dereference. Pointer-based solutions either attach this meta-data directly to a pointer (fat pointers) or store it disjointly in a shadow memory region. Using disjoint meta-data has the advantage of achieving binary compatibility with unsafe libraries, while having to maintain the shadow memory region.

SafeCode is an object-based approach and requires minimal changes to the source code of the program. SafeCode uses a points-to analysis based technique called Automatic Pool Allocation to reduce the overhead of memory safety checks [LA05]. For every node in the points-to graph a separate partition of memory is introduced. First a pointer may never point outside of its assigned memory pool. Second by pooling memory allocation the target object associated with a pointer can be looked up more efficiently. When performing pointer arithmetic they introduce checks that ensure that the underlying object is still the same [DA06]. Furthermore pool allocation with type-homogeneous pools provides a natural way to ensure that dangling pointers never violate type safety guarantees. SafeCode achieves this by preventing cross-pool memory reuse [Dhu+03; DKA06]. A general problem with object-based approaches is that out-of-bounds pointers are allowed in the C language, as long as they are not dereferenced. This is a pattern that is common in many C programs and needs special handling in object-based approaches. Furthermore object-based checks have trouble dealing with sub-object memory corruption, such as overflows inside of a `struct` or inside of arrays [Nag12].

Some pointer-based proposals modify the C languages and restrict the problematic parts of the language. For example CCured [NMW02] and Cyclone [Gro+02; Hic+04] introduce a strong type system into the C language. Based on the strong type system they can decide which pointers are safe to use and which must be explicitly checked during runtime. They furthermore tackle temporal safety by introducing some form of automatic memory management. The disadvantage of these approaches is that they introduce new language constructs and therefore require source code modifications.

SoftBound [Nag+09] is a highly compatible pointer-based approach for ensuring spatial safety. For every pointer SoftBound keeps track of a base and a bound value. This base and bound values are updated during pointer arithmetic. When a pointer is dereferenced the base and bound values are retrieved and it is checked whether the pointer still points into the associated object. Furthermore, when the address of a field inside of a structure is taken, SoftBound shrinks the base and bound value to the field. This allows prevention of sub-object spatial memory corruption. Compiler enforced temporal safety for C (CETS) [Nag+10] is a scheme to ensure temporal safety, given that the program is free of spatial memory safety violations. CETS introduces a allocation key and a lock value for each allocation, which are propagated during pointer arithmetic. To detect dangling pointers CETS inserts checks that verify that the key and lock value are the same. The lock value is set to an invalid value during deallocation. Furthermore, a hash-table is used to keep track of freeable pointers and detect invalid parameters to calls to `free`.

MemSafe [SB13] is a hybrid approach that combines pointer-based checking, similar to SoftBound, and object-based checking. To ensure spatial memory safety a pointer-based disjoint meta-data approach similar to SoftBound is used. MemSafe models temporal safety violations as spatial memory safety violations. On deallocation the pointer and object-based meta-data is set to an invalid value. This cannot be propagated to pointers to fields inside structures. Therefore to ensure complete temporal safety MemSafe also keeps track of object-based meta-data, which allows a lookup from pointers to fields to the object meta-data. To avoid duplicate bounds-checks MemSafe relies on safety analysis to allow it to safely eliminate checks.

### 2.3.3 Probabilistic Defenses

Probabilistic defenses are mechanisms that introduce some form of randomness into the execution of a program, while preserving the semantics of the original program. This makes exploits non-deterministic and therefore harder to construct. Probabilistic defenses are by nature not a complete protection and break down as soon as the randomized values can be inferred by the attacker. While Probabilistic defenses cannot offer complete protection, they have several advantages. Many of the proposed probabilistic defense mechanisms have a low performance overhead and do not suffer from compatibility issues. This makes them easy to implement and deploy. Furthermore the analysis of the security guarantees of an probabilistic defense is rather easy: one measures the provided entropy in the randomized values. This assumes that the randomized value cannot be inferred other than by brute-force guessing. Some probabilistic schemes also feature a tweakable security parameter, that allows to configure the introduced entropy. Probabilistic defenses can be applied in many places during the execution of a program. ASLR randomizes pointer values. Software diversification is used to create unique binaries, to hide low-level details of the binary from the attacker. Other schemes introduce blinding by using the XOR operation with a secret blinding values.

#### Pointer Randomization

One of the most wide-spread defense mechanisms is ASLR. ASLR randomizes the address layout of a process during the process creation. Position independent sections are mapped at randomized virtual addresses. This means that an attacker does not know the address of the stack or the heap. This thwarts simple code injection attacks by having the attacker guess the address of the injected code.

Attacks against ASLR have been widely studied, both by exploit writers and academia. One of the first attacks against ASLR attacked the low available entropy on 32-bit address space. For practical reasons ASLR implementations cannot randomize the full 32-bit of an address and provide at most an entropy of 25-bit and for some types of mapped memory even less. A brute-force attack is therefore possible on 32-bit systems [Sha+04]. Nowadays even mobile devices feature 64-bit address space, making ASLR much more effective on these platforms.

Practical exploits use non-randomized sections to leak or infer addresses of the randomized sections. For example the code section of an executable cannot always be mapped to a randomized addresses, because it is common to compile code to use constant addresses for jump and call targets. Dynamically linked libraries are compiled in a position-independent way, because their position is not known at compile time. An attacker can still use the main executable for code-reuse attacks. Depending on the binary, there might be enough gadgets available for Turing-complete computation. Furthermore an attacker can also call all the library functions called from the main executable. With this capabilities attackers can construct exploits that first leak the randomized addresses and update their payload to use the leaked addresses [Str+09]. Usage of position-independent code for main executables becomes more frequent, to improve the effectiveness of ASLR. This randomizes all mapped code pages and requires an information leak of a code pointer, before constructing an attack.

A big problem for ASLR is when an attacker can retry exploits without a re-randomization of the targeted process' address space. This is for example common on forking network servers. A child process inherits the address space layout from the parent process. Because of performance reasons, the address space is usually not re-randomized. An attacker can therefore try her exploit with different addresses until it works. The blind ROP technique introduced in [Bit+14] allows an attacker to create a working ROP exploit even when the attacked binary is not known. They perform a brute-force attack to locate suitable gadgets to execute a small ROP chain. The only assumption is that the server process forks for each connection and does not re-randomize the address space. A similar technique can also be applied to some client applications. The Javascript engines in web browsers often allow an attacker to execute a crashing exploit in a thread, without crashing the whole browser process [Gaw+16b]. Using such crash-resistant primitives an attacker can de-randomize ASLR or perform blind ROP attacks.

### Code Randomization

Because of the relatively low entropy in ASLR more fine-grained randomization schemes have been proposed. Address Space Layout Permutation [Kil+06] re-orders the functions in a binary during program loading. XIFER randomizes the order and position of basic blocks [Dav+12b]. Compile and load-time diversification of the executable code already provide a significant problem for attackers. Simple exploits do not work anymore. Attackers have to resort to information leaks to gain as much information about the executed code as possible. With advanced code-reuse techniques such as blind return oriented programming (BROP) and just-in-time return oriented programming (JIT-ROP) attackers can defeat even fine-grained randomization schemes.

As a reaction to the JIT-ROP attack, several different approaches have been proposed to counter information leaks of randomized code. Multivariate execution is a technique where the same input is concurrently fed to the same program, but with different randomization. If the output of the programs differ, then an information leak happened. Besides the high overhead of execution a program twice, there are many practical problems with eliminating all sources of non-deterministic behavior from a program [Gaw+16a]. Shuffler [Wil+16] is a scheme that continuously re-randomizing code sections during execution. An attacker is forced to complete a JIT-ROP attack before the re-randomization occurs. This approach imposes a high overhead for the continuous code randomization.

Other solutions try to prevent information leaks in the first place. For example [Bac+14] proposed using executable, but not readable, memory for code pages. Code is typically not read using other instructions, but is only read by the CPU during instruction fetching. For optimization purposes compiler sometimes store constants directly into the code segment, for example for instruction pointer relative loading of constants. This breaks the assumption of execute-only memory, that code pages are never read. To use the technique without access to source code workarounds need to be built in, that also give an attacker opportunities to attack the system.

In contrast to execute-only memory, destructive code reads [TSS15] do not prevent information leaks of code pages, but render leaked code unusable.

When code is read, it is destroyed afterwards by overwriting it with some instruction, that triggers a program abort. An attacker cannot execute an instruction anymore after it has been read by an information leak of the attacker. Destructive code reads have been undermined by code inference attacks [Sno+16]. If code has been mapped twice into the address space of a program, the attacker can read one copy and execute the other. Even if this is not possible, then an attacker can leak the first couple of basic blocks to identify them. The remaining instructions can then be used by the attacker for code-reuse attacks.

Furthermore fine-grained randomization cannot defeat function-level code reuse attacks. The semantics of a function must be preserved by code-randomization schemes. For example on 32-bit `x86`, where parameters are usually passed on the stack, an attacker can perform return-to-libc attacks. First the attacker needs to disclose the addresses of the used functions, to bypass ASLR. An attacker can leak function pointers, return address or jump tables to locate existing functions. Fine grained code randomization does not offer any advantage over plain ASLR against function-level code reuse. The majority of architectures, such as `x86_64`, ARM, MIPS, do not pass function arguments on the stack. An attacker must therefore reuse existing code to set the function arguments and fine-grained code randomization does impose a significant challenge to the attacker. Furthermore many code-randomization schemes can partially re-randomize code on `fork` [Wil+16], so blind ROP becomes infeasible. JIT-ROP attacks are still feasible if the arbitrary read can be triggered without crashing the process. JIT-ROP itself only leaks at mapped addresses, so it does not crash the process.

COOP is an attack technique that bypasses all the existing code randomization schemes and also techniques that prevent code leakage. The attacker must only leak the addresses of the C++ `vtables`, which can be achieved by leaking the contents of objects. Furthermore to perform a COOP attack, the attacker does not need to know the exact code, but only the semantics of the C++ virtual methods and the layout of the C++ classes. As a response to attacks, that disclose code locations, several schemes have been proposed to hide code references. Oxymoron [BN14] hides all direct references to other code pages, by turning direct branches into indirect branches. This hinders JIT-ROP attacks, which use direct branches to discover further code pages. Oxymoron does not protect against indirect code disclosure by leaking code pointers. As shown in [Dav+15] indirect code disclosure is sufficient to create working exploits. Readactor [Cra+15b] hides return addresses and function pointers behind execute-only trampolines.

To mitigate COOP attacks Crane et al. proposed to randomize code pointer tables, such as `vtables` [Cra+15a]. They hide entries in the code pointer tables behind execute-only trampolines. This prevents an attacker from disclosing the address of the randomized virtual methods. This way an attacker cannot inject fake `vtables`. To prevent an attacker from guessing the correct index into a `vtable`, they introduce "booby traps" into code pointer tables, which are invalid indices that upon use trigger a program abort or crash. An attacker would need to guess the randomized index into a `vtable`, but is very likely to hit one of the booby trapped indices.

#### Blinding Schemes

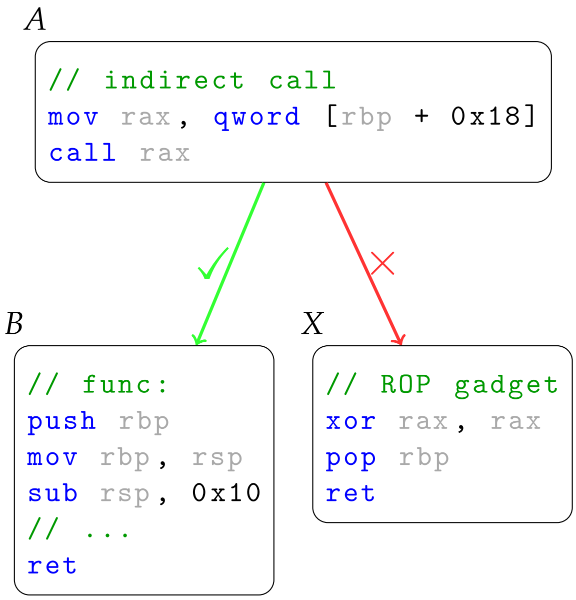
Many schemes introduce secret random values, by blinding data. This is usually done by applying th XOR operation to the original value and a secret random value. Blinding schemes are often used to mitigate shortcomings of code randomization schemes or to achieve low overhead protections. For example PointGuard [Cow+03] protects all pointers. Many code randomization schemes blind return addresses, to mitigate code pointer leaks [Wil+16]. PaX RAP [Tea15] introduces an additional layer of probabilistic protection on top of a deterministic CFI defense mechanism. Instruction Set Randomization is a technique that blinds all instructions inside the code segment of a program [KKP03]. We discuss general non-control data blinding schemes, a probabilistic form of data-flow integrity (DFI), in Chapter 3.

### 2.3.4 Control-Flow Integrity

CFI is a mitigation mechanism against control-flow hijacking attacks and was first introduced by Abadi et al. [Aba+05]. CFI is a deterministic protection that limits the capabilities of an attacker. It both defines a security policy and enforces this policy by introducing additional checks in the protected program. The CFI security policy is: during execution a program must adhere to its CFG, which has been determined before the execution. During typical control-flow hijacking attacks, such as ROP, there are many control-flow transfers that violate this policy. Typical ROP attacks consist mostly of control-flow transfers that are not part of the static CFG. However an attacker that is aware of the restrictions that CFI impose, can still operate within the CFG. The assumption is that it is unlikely that an attacker is able to construct a meaningful exploit with the imposed restrictions.

CFI considers a threat model that assumes correctness of the processor and an absence of physical attacks. Furthermore it is assumed that the code protected by CFI cannot be modified, i.e. is mapped read-only. CFI also assumes that data is non-executable, otherwise an attacker could inject correctly code bypassing CFI as data. CFI must be considered a defense only against code-reuse attacks and must be combined with proper enforcement of memory protections. With this assumptions the attackable control-flow transfers are those, that happen because of indirect branching instructions. Direct branches can only have one target in the CFG and therefore use a constant address as target, which is encoded in read-only instructions. CFI checks can be omitted for direct branches. Indirect branch instructions direct control-flow to a variable code pointer, which can be corrupted by an attacker. Examples for indirect branching instructions on x86 are indirect jump, call and return instructions.

**Forward-edge CFI** is considering the protection of indirect call and jump instructions. Basic block *A* in Figure 2.4 shows the assembly instructions for performing an indirect call. First the target code pointer is loaded from the stack into the `rax` register. Then `rax` is used as parameter to the `call` instruction. A legitimate transfer would be to a function, that fits the CFG. The basic block *B* is a valid call target, as it contains the function prologue of a function within the CFG. On the other hand a transfer to $X$ clearly violates CFI, as it would jump into the middle of a function. To enforce CFI a check must be inserted in front of the `call` instruction that verifies the call target.



```
// indirect call
mov rax, qword [rbp + 0x18]
call rax
```

```
// func:
push rbp
mov rbp, rsp
sub rsp, 0x10
// ...
ret
```

```
// ROP gadget
xor rax, rax
pop rbp
ret
```

Figure 2.4: Forward-edge CFI: On the left a valid transfer and on the right a malicious one.

**Backward-edge CFI** is concerned with protecting returns from a call. Figure 2.5 shows an example for a backward control-flow transfer via a return instruction. The `main` function calls `func` and returning from basic-block $B$ to $A$ is a valid control-flow transfer because the return address points to a instruction that is preceded by a call to `func`. On the other hand a return to a ROP gadget at $X$ is an invalid transfer to a ROP gadget, violating CFI. To enforce CFI a check must be inserted before the return, that ensures that the return address is a valid return target.

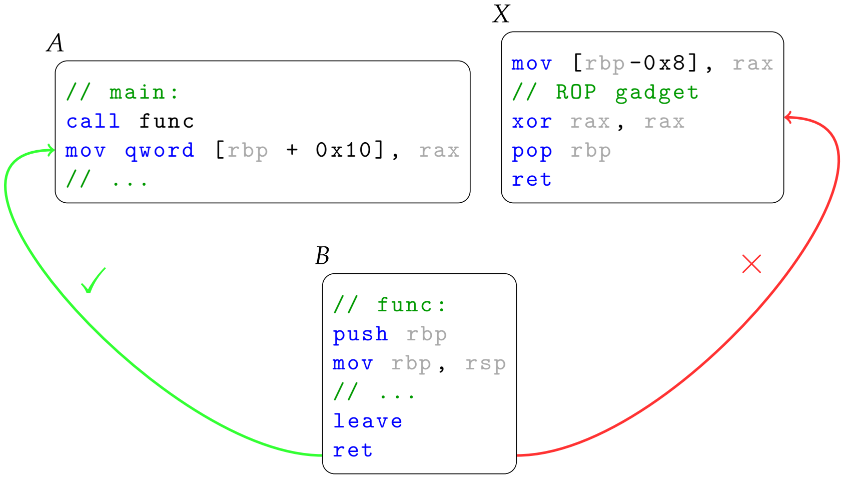


Figure 2.5: Backward-edge CFI: On the left a valid return and on the right a malicious one.

Returns addresses are code pointers that occur much more frequent than other code pointers. A return address is saved on the stack for every function call, which means that return addresses are also very likely to be targeted by an attacker. Unfortunately checking every return address is also very expensive. Many CFI implementations therefore use shadow/split stack setups or probabilistic defences to protect them from an attacker and omit explicitly checking CFI for return addresses [Aba+09; Dav+14].

A CFI implementation as proposed by Abadi et al. [Aba+05] induces a relatively high overhead on the protected program. To decrease the performance impact of CFI, implementations often use a more coarse-grained security policy. For example as can be seen in Figure 2.5 the instruction at a return address must always be preceded by a `call` instruction. A very coarse-grained backward-edge CFI policy could require that valid targets for returns are only those addresses, which are preceded by some call instruction. kBouncer is a defense mechanisms against ROP, that implements this policy [Pap12]. According to Schuster et al. [Sch+14] this policy is too coarse-grained to prevent exploits by an attacker that is aware of the restrictions.

```
typedef struct msg {
  char str[64];
  int (*print_fn)(char*);
} msg_t;

int system(char* cmd) { /* ... executes command */ }
int log_stdout(char* s) { /* ... */  }
int log_file(char* s) { /* ... */ }

int main(int argc, char* argv[]) {
  // ...
  msg_t msg = { .str = {0, }, .print_fn = log_stdout};
  size_t isz = strlen(user_inp) + 1;
  memcpy(msg.str,
         user_inp,  /* <-- attacker controlled */
         /* this is a buggy length check. */
         isz < sizeof(msg.str) ? isz : sizeof(msg));
  // the attack is able to corrupt the function pointer
  // msg.print_fn and can achieve aribtrary code
  // execution here:
  msg.print_fn(msg.str);
  // ...
}
```

Listing 2.5: Most CFI implementations can be bypassed in this example.

Even the most accurate current CFI implementations suffer from inaccuracy resulting from the over-approximation of the set of allowed call targets. Listing 2.5 shows an example where an attacker can manipulate a function pointer via a spatial memory corruption vulnerability. In this case the attacker can overwrite the function pointer in the msg_t structure. The first argument to the print_fn call is also an attacker controlled string. For an successful exploit the attacker can corrupt the print_fn pointer to point to the system function. On the last line the call to print_fn is executed and the attacker gains the ability to execute arbitrary system commands. For this example we assume that this really is a CFI violation and system is never assigned to print_fn anywhere in the code.

Coarse-grained CFI implementations such as Windows Control-Flow-Guard or Intel Control-flow Enforcement Technology will only prevent calls to code that is not the beginning of the function. Since in this case the attacker performs code-reuse on the function level, no CFI violations would be reported [Dav+14]. Even more fine-grained CFI implementations such as the forward-edge CFI included in the clang/llvm compiler [Tic+14] or the forward-edge protection in `Pax RAP` [Tea15] would not prevent this exploit. The reason is that both systems do not perform extensive points-to analysis. Both solutions use an inexpensive type-based analysis to determine which functions can be called. The function signature of the `print_fn` function pointer is exactly the same as the `system` function signature so these solutions determine that a control-flow transfer to `system` is legitimate. As a result an attacker is able to bypass CFI in this case and transfer control to the `system` function and gain access to a system shell.

Practical CFI implementations suffer from over-approximation and coarse-grained security policies. Many CFI implementations have been proposed [Aba+05; Dav+12a; Tic+14; PBG15] and ways to bypass them have been found [Con+15; Gok+14]. Because of the difficulty of evaluating the security guarantees of CFI, an analysis of a theoretical fully precise static CFI solution has been performed in [Car+15]. They consider several case studies of real-world exploits and check whether the vulnerabilities can be abused for meaningful attacks without violating the fully precise static CFG of the targeted program.

CFI solutions without the use of shadow stack, allow easy traversing of the CFG by abusing so-called dispatcher functions. Dispatcher functions are functions that are able to overwrite their own return address and are called from many locations. Examples for dispatcher function in [Car+15] are: `memcpy`, `strcat`, `printf` and `fputs`, but many more suitable functions exist. The dispatcher function can be used to return to a large set of destinations, all those that are preceded by a call to the dispatcher function. Usage of a shadow stack to protect return addresses is therefore crucial for CFI solutions [Car+15; Con+15].

Even with the presence of a shadow stack, an attacker can still corrupt other code pointers to hijack control-flow and traverse the CFG. Listing 2.6 shows a short snippet of code, in which even fully precise static CFI cannot prevent

```
int system(char* cmd) { /* ... executes command */ }
int deny(char* cmd) { puts("Denied!"); return -1; }

int main(int argc, char* argv[]) {
  bool is_admin = false;
  // ...
  char buf[128];
  int (*fptr)(const char*);
  // ...
  if (is_admin) {
    fptr = system;
  } else {
    fptr = deny;
  }
  // fptr can point to both system and deny
  // ...
  // calls to gets are vulnerable to buffer overflows
  gets(buf);
  // the attacker can corrupt the fptr variable
  // ...
  fptr(some_string); // <-- control-flow hijack
  // ...
```

Listing 2.6: Even fully precise static CFI can be bypassed here.

an exploit. A function pointer `fptr` can be corrupted by a buffer overflow. In case the user is an administrator, she is allowed to execute arbitrary commands and therefore `fptr` is set to the `system` function. Otherwise `fptr` points to the `deny` function, which ignores requests to execute commands. We assume that an attacker does not have administrator privileges by default. In the precise static CFG both the `system` and `deny` functions are in the set of allowed targets of the indirect call via `fptr`. The attacker can abuse the buffer overflow to manipulate the `fptr` function pointer and elevate her privileges, while staying within the valid CFG. In this example the `is_admin` flag cannot be manipulated, but would be an equally interesting target for a data-only attack.

In three of the six case studies of real-world vulnerabilities in [Car+15] it was possible to craft attacks that achieved arbitrary code/command execution without violating the CFG. This shows that in many real-world programs, the CFG is permissive enough such that even a severely restricted attacker can create valuable exploits. It is therefore unlikely that CFI solutions alone offer enough protection against an sophisticated attacker.

### 2.3.5 Code Pointer Protection

As can be seen in previous sections, to launch a control-flow hijack an attacker must overwrite a code pointer. Naturally defenses came up that prevent or detect corruption of code pointers. First defenses were developed for specific code pointers and then generalized to all code pointers.

A widely deployed defense mechanism are stack canaries. A canary is a secret random value that is placed right in front of the return address. Before returning from a function, the function checks whether the canary has been modified. If it was modified, it is very likely that also the return address has been corrupted. This way linear stack-based buffer overflows can be detected [Cow+99]. Stack canaries are widely deployed and available in production compilers, such as `gcc`, `llvm` and the Microsoft Visual C/C++ compiler.

Shadow or split stack setups save the return address on a different stack to ensure the integrity of return addresses. Read-only relocations (RELRO) is a technique that protects the GOT, which is a code pointer table used to resolve dynamically linked functions. The dynamically linked functions are resolved during load-time, instead of lazily during runtime, which makes it possible to map the GOT read-only into the address space.

Cryptographic CFI [Mas+15] is a scheme that protects all code pointers by computing a message authentication code (MAC) over the pointer and the pointer location. While the technique is called CFI, it is actually a code pointer protection scheme. The CFG is not required to be computed beforehand to implement this scheme and it detects code pointer corruption.

Code pointer integrity (CPI) [Kuz+14] is a generalization of all code pointer protection schemes. CPI introduces memory safety for a subset of the variables in a program: code pointers. Furthermore they extend this recursively to all pointers pointing to code pointers, for example pointers to C++ virtual method tables. All code pointers and pointers to code pointers are moved to a safe region. All accesses into the safe region are instrumented to include additional checks to ensure memory safety inside the safe region. This effectively prevents control-flow hijacking attempts, since the attacker cannot corrupt code pointers anymore.

This scheme is still vulnerable to data-only attacks. So an attacker can try to use memory corruption of data pointers and variables in the unsafe region, to corrupt memory in the safe region. Therefore the safe region must be isolated from the unsafe region. Their prototype for `x86` uses segmentation to isolate both regions. The prototype implementation for `x86_64` uses ASLR to hide the safe region from the attacker. This hiding based approach turned out to be not effective against data-only attacks [Eva+15]. A software fault isolation (SFI) based approach has also been proposed, which does not suffer from this problem, but induces higher overhead.

# 3 Data-Flow Integrity Schemes

Data-flow integrity (DFI) is a mitigation mechanism that enforces limitations on the data flow during execution of a program. DFI ensures that data is never written to or read from unintended locations. The policy, which is enforced by DFI implementations, is based on the data-flow graph that static analysis computes before program execution. To counter performance problems, some variants of DFI enforce more coarse-grained policies. DFI was first introduced in [CCH06]. They use reaching definitions and points-to analysis [ASU86] to compute the static data-flow graph. The program is then instrumented to enforce the static data-flow graph. The static analysis must return an over-approximated result to preserve the soundness of the analysis. The static analysis can add edges to the data-flow graph, which are impossible during executions of the program. Enforcing the integrity of data flow prevents many memory corruption exploits.

The concept and the first implementation of DFI was introduced by Castro, Costa, and Harris [CCH06]. A variant of DFI with lower overhead and less security guarantees is write-integrity testing (WIT) [Akr+08]. WIT does not enforce integrity for load instructions. Instructions that write data are far less common than instructions that read them. By only instrumenting write instruction, WIT achieves lower overhead, sacrificing protection against some information leaks. With data space randomization [BS08] and data randomization [Cad+08] two probabilistic implementations of DFI have been proposed concurrently. Both use random values to blind data.

In this chapter, we describe the concept and implementations of DFI. We discuss the original DFI proposal and probabilistic implementations of DFI, data randomization, and data space randomization. We then describe how WIT implements a subset of DFI, write integrity.

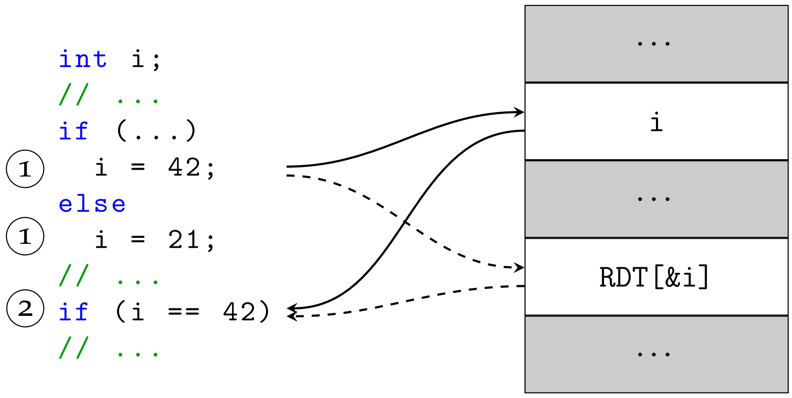
## 3.1 Data-flow Integrity

A central observation regarding memory corruption exploits is that they usually start with an unintended data flow. For example, a buffer overflow stores data not in the intended object, but also in an adjacent object. The out-of-bounds write, triggered by the buffer-overflow, is a data flow to an object, which is not intended by the developer. Whether the data flow to or from an object is valid can be checked by statically determining what objects may be written or read by certain instructions. Data flows, which an attacker can corrupt, are those that use pointers to refer to memory locations. Similar to control-flow integrity (CFI), where only indirect branches need to be checked, DFI is only concerned with instructions that use pointers. For example, instructions that perform arithmetic on values stored in registers can be considered as safe in the context of DFI. To violate data flow between registers, an attacker would have to hijack control-flow, which DFI mitigates by also detecting invalid data flows to code pointers.

DFI uses a reaching definitions analysis to compute the static data-flow graph. A definition of a memory location is an instruction, that writes to the location. A user of a memory location is an instruction, that reads from it. First, the sets of defining and using instructions are computed. For intra-procedural analysis, this is achieved by traversing the single static assignment (SSA) form of the analysed program. The inter-procedural analysis is more involved, as points-to analysis is required to decide which objects can be accessed. A points-to analysis outputs a directed graph, where a node corresponds to a set of objects, and an edge represents the fact that a pointer might point to a set of objects. Since pointers can point to other pointers, which are also objects, we have a points-to graph rather than a simple mapping between pointers and sets of objects. DFI, as proposed by [CCH06], uses a field-insensitive inclusion-based points-to analysis also called Andersen-style points-to analysis. All proposed DFI schemes use a context-insensitive points-to analysis, which identifies objects by their allocation site.

DFI keeps track of what instruction defined a memory object. For every user of a variable, DFI needs to check whether the definition of the variable is reachable from the user in the static data-flow graph. The way DFI keeps track of this information is to store an identifier of the last instruction, that defined the value. The identifier is stored in a table, called the runtime definitions table (RDT). Every store instruction is instrumented to record its identifier in the RDT. Every read instruction is instrumented to verify that the last instruction that defined the value is actually in the set of instructions, which are allowed to define the value. DFI extracts the set of allowed definitions from the static data-flow graph. DFI does not consider single definitions. It collapses all definitions with the same users into one equivalence class.

Figure 3.1 shows a simple snippet of C code and the DFI operations. The value `i` is defined by the instructions ① and it is used by the comparison at ②. The instructions at ① define only one value and therefore the users are also the same, and DFI merges them into an equivalence class. When the value `i` is defined, the RDT is updated accordingly. The read instruction then retrieves the value from the RDT and verifies that the last write was indeed by an instruction in the set of instructions ①.



```
    int i;
    // ...
    if (...)
①     i = 42;
    else
①     i = 21;
    // ...
②   if (i == 42)
    // ...
```

Figure 3.1: DFI: Instrumented writes update the RDT and instrumented reads verify the contents of the RDT.

This way memory corruption attacks can be detected. For example, consider a buffer overflow that overwrites an adjacent boolean variable. The store instruction that causes the buffer overflow would update the RDT entry of the variable accordingly. Upon use of the variable, the RDT entry does not match the allowed set of instructions and DFI would terminate the process.

A major problem with DFI is the high instrumentation overhead. The original DFI proposal [CCH06] reported a runtime overhead between 44% and 103% in the Spec 2000 benchmark. Furthermore, the memory overhead is about 50% for DFI. DFI needs to instrument each store instruction to additionally update the RDT. Furthermore, the operand of a store instructions needs to be checked, such that the store cannot write into the RDT. The overhead for each instrumented load instruction is another load instruction to fetch from the RDT, the validation of the value in the RDT, and a branch instruction to abort in the case of a DFI violation.

To counter the performance issues, probabilistic versions of DFI were proposed. Data randomization [Cad+08] and data space randomization [BS08] were proposed concurrently and both propose probabilistic DFI variants. Data randomization is implemented as a compiler transformation, which induces a runtime overhead between 0% and 27%. Data space randomization is implemented using source code rewriting and has a runtime overhead between 4% and 28%.

Both data randomization and data space randomization offer a probabilistic version of DFI by blinding all allocated objects with the XOR operation and a secret random mask. They instrument all load and store instructions to apply the random mask. If an instruction violates DFI, the wrong mask would be applied and the value would be garbage. For example, an attacker abuses a buffer overflow of a buffer $b$ to overwrite a return address. The buffer $b$ would be masked with $m_b$ and the return address $r$ with $m_r$. When the corrupted return address is read, the actual value would be $r' \oplus m_b \oplus m_r$. The first mask $m_b$ is applied when the user data is written into the buffer. The second mask $m_r$ is applied when the return address is read. If an attacker has knowledge about the random masks, she would still be able to conduct the attack. The attacker would overwrite the return address with $r'' = r' \oplus m_b \oplus m_r$. The instrumented code would apply $m_b$ and $m_r$, leaving $r'$ as the used return address.

To make sure that the correct mask is applied to an object, all definitions and all uses of an object must apply the same mask. If an object is accessed through a pointer it must also be checked whether the points-to set contains other objects. All objects, which have at least one use or one definition in common, must be blinded with the same mask. This creates equivalence classes that are computed based on the points-to graph. The equivalence are the same as in DFI.

## 3.2 Write Integrity Testing

WIT is a performance optimized variant of DFI, that sacrifices read integrity for performance. WIT as implemented by Akritidis et al. [Akr+08] induces a runtime overhead between 7% and 25%. The memory overhead of WIT is reported to be approximately 12.5%, because of the more compact shadow memory representation. The definition of the static security policy is very similar, except that WIT does not enforce integrity on load instructions. WIT uses a points-to analysis to compute its policy. In contrast to DFI, WIT is not concerned with users of an object. To enforce write integrity, WIT first computes the points-to graph for each store instruction. WIT assigns a color to each store instruction, such that all objects written by the instruction are colored the same way. If two different store instructions can write to the same object, then both store instructions are assigned the same color. WIT computes equivalence classes of stores and objects. Each store instruction is allowed to write to any of the objects in the associated set of objects.

Figure 3.2 shows an example of the color computation of WIT. First, the points-to graph is computed. The pointers used by store instructions are analysed to find the set of objects that may be referenced by the pointer. The store instruction, which uses that pointer, is assigned the set of objects writable through that pointer. In Figure 3.2, the store instruction $X$ can write to the objects $A$ and $B$. The store instruction $Y$ can write to $C$ and $B$. Both store instructions are assigned the same color because an object can only have one color assigned at a time. The store instruction $Y$ can also write to object $A$, although the static analysis could determine that such a data flow would be illegal. On the other hand, only the store instruction $Z$ writes to the objects $D$ and $E$. WIT can enforce that no other store instruction writes to those objects. An attacker can still exchange the objects $D$ and $E$ if she manages to corrupt the pointer %6 used in the store instruction $Z$.

To illustrate the protection that is offered by WIT an example of a data-only attack thwarted by WIT is shown in Figure 3.3. The figure shows the

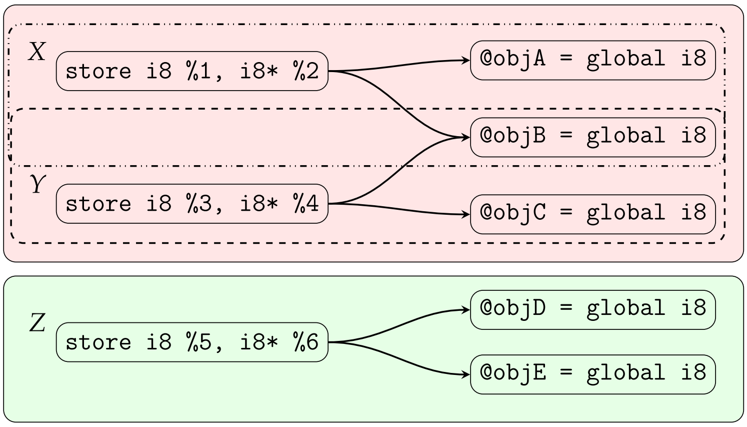
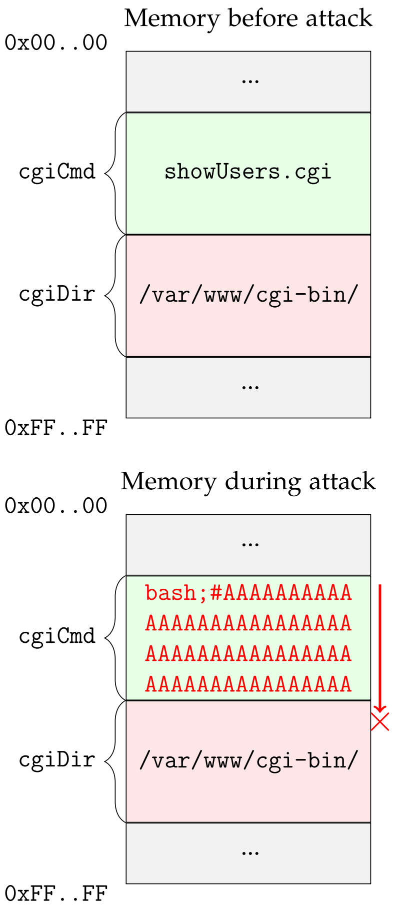


Figure 3.2: Example coloring of WIT stores (in LLVM IR).

same snippet of C code, as shown in Figure 2.3 in Chapter 2. The attacker again tries to abuse the buffer overflow from `cgiCmd` to `cgiDir` to achieve arbitrary command execution, but WIT will prevent the attack. The WIT colorings are equal to the enforced equivalence classes. Since `cgiCmd` is only written by `ProcessCGIRequest` and `cgiDir` is only written by `SetCGIDir`, both can be cleanly separated into equivalence classes that do not overlap. We assign the store instruction in `ProcessCGIRequest` and `cgiCmd` the color green. The store instruction in `SetCGIDir` and `cgiDir` are assigned the color red. The WIT instrumentation colors the objects during program startup. The assignments in line 14 and 25 are instrumented to check that the objects they are writing match their assigned color. When the attacker uses the assignment in line 14 to write out-of-bounds of the `cgiCmd` object and into the `cgiDir` object, WIT catches the color mismatch and aborts execution of the program.

The lack of read integrity enforcement results in WIT tolerating certain kinds of information leaks. For example, the infamous Heartbleed [Tea] vulnerability would be left unmitigated by WIT. Heartbleed was a rather simple heap-based buffer over-read. Since WIT does not enforce any restrictions on load instructions, a vulnerability like Heartbleed cannot be detected at all. Other information leaks require the corruption of a pointer. They can be detected by WIT when the attacker corrupts the pointer.



```
// written only by ProcessCGIRequest
// WIT color: green
char cgiCmd[64];
// written only by SetCGIDir
// WIT color: red
char cgiDir[64];

void ProcessCGIRequest(char* curCmd) {
  // curCmd is untrusted input
  size_t i = 0;
  // buffer overflow here
  while (i < strlen(curCmd)) {
    // WIT color: green
    cgiCmd[i] = curCmd[i];
    i++;
  }
  // system(strcat(cgiDir, cgiCmd))
  ExecuteRequest(cgiDir, cgiCmd);
}

void SetCGIDir(char* newDir) {
  size_t i = 0;
  while (newDir[i] && (i < 64)) {
    // WIT color: red
    cgiDir[i] = newDir[i];
    i++;
  }
}
```

Figure 3.3: Buffer overflow between two data variables is prevented by WIT.

# 4 Write Integrity Testing Implementation

We implemented a prototype of write-integrity testing (WIT) within the LLVM [LA04] compiler framework. For our prototype, we stayed as close as possible to the original implementation of WIT [Akr+08]. Based on our prototype, we evaluated the WIT mitigation mechanism. In this chapter, we describe the various implementation choices and lessons learned while implementing our LLVM-based WIT prototype (LLWIT).

In contrast to the original proposal of WIT, we omit several enhancements, that are not relevant for assessing the security guarantees offered by WIT. The original WIT performed write-safety analysis to determine, whether a store instruction can be considered safe. The write-safety analysis tries to find a proof, that the pointer used by the store instruction, will never go out-of-bounds for any object in the points-to set. Instrumentation of safe store instructions can be omitted. This significantly reduces the overhead of a WIT implementation [Akr+08]. We omit to perform an extensive write-safety analysis, as this is primarily a performance improvement. We use the LLVM `mem2reg` pass to promote stack allocations to LLVM intermediate values. We can skip instrumentation of stores to primitive types, that are allocated on the stack. Our prototype then instruments all remaining store instructions. Furthermore, we omit the special handling of dynamically linked libraries. We simply link all relevant functions statically. We do not need to build a special instrumentation for libraries.

The original WIT implementation also included a forward-edge control-flow integrity (CFI) scheme, that uses the same points-to analysis and colorings. For each indirect jump, they computed the points-to set and introduced coloring in the shadow memory. We omit this from our implementation, as there is already a CFI implementation in LLVM and it does not impact the protection against data-only attacks. The CFI scheme included in LLVM is less precise, as it uses a inexpensive type-based alias analysis for computing the valid call targets. We consider it out of scope for this thesis to combine the more precise points-to analysis from WIT with the CFI implementation in LLVM.

Our compilation pipeline is built in the following way: First, we compile all source files to LLVM bitcode. We then use the bitcode linker to create one single bitcode file. Since WIT requires a whole-program points-to analysis, we must perform analysis on the linked bitcode file. After the points-to analysis, we start the instrumentation pass. The last step links in our support library and creates an executable.

In our prototype, we use an Andersen-style points-to analysis, similar to the original WIT proposal. We adapted a publicly available implementation[1] of the Andersen points-to analysis for the LLVM framework. We improved modeling of external libraries in the points-to analysis. Furthermore, we had to adapt the points-to analysis to work, when the allocator functions, like `malloc` and `free`, are statically linked functions.

When the points-to analysis fails for a given pointer, it returns a universal node, which represents the set of all objects. For WIT, this is a disastrous result, as it would mean that all objects and stores would coalesce into the same color. Our WIT prototype would still protect target specific data, such as return addresses and saved registers, but would otherwise be rather useless. This situation can be handled by skipping instrumentation of the store, where the points-to analysis has failed. We sacrifice security guarantees, but we can continue without having the degenerate case of one color and without introducing compatibility issues.

Listing 4.1 shows an excerpt of LLVM intermediate code, that is analysed by our WIT prototype. The code is an excerpt from the example in Figure 3.3 compiled to LLVM intermediate code. We have the two global variables `cgiCmd` and `cgiDir`, both character arrays. In the function `ProcessCGIRequest` we have one store instruction. The `getelementptr` instruction is used to retrieve an element from a compound data structure.

[1] https://github.com/grievejia/andersen

```
@cgiCmd = common global [64 x i8]
@cgiDir = common global [64 x i8]

define void @ProcessCGIRequest(i8* %msg, i32 %sz) #0 {
  ;  ...
  ; cgiCmd[i] = msg[i];
  %15 = getelementptr inbounds [64 x i8],
                           [64 x i8]* @cgiCmd,
                           i32 0, i64 %14
  ; points-to-set(%15) = {@cgiCmd}
  store i8 %12, i8* %15, align 1
  ;  ...
}

define void @SetCGIDir(i8* %newDir) #0 {
  ;  ...
  ; cgiDir[i] = newDir[i];
  ; points-to-set(%17) = {@cgiDir}
  %17 = getelementptr inbounds [64 x i8],
                           [64 x i8]* @cgiDir,
                           i32 0, i64 %16
  store i8 %14, i8* %17, align 1
  ;  ...
}
```

Listing 4.1: Points-to analysis and derived coloring in LLVM IR

In this case, it computes a pointer into the array `cgiCmd`. The points-to analysis determines that the pointer %15 can only point to the `cgiCmd` object. We can then partition the objects and store instructions into two different colors.

After the points-to analysis and the color computation, the next step is to instrument the code to update and check the coloring information. We store the color metadata in a shadow memory region. We represent each color with a numeric value. We instrument all store instructions to fetch the color of the accessed object from the shadow memory and compare it with the color we assigned to the store instruction. If the comparison succeeds we continue normal execution and otherwise branch to error reporting code.

We use a shadow memory representation similar to the one use by the AddressSanitizer included in LLVM [Ser+12]. We store one byte of color information for every 8-byte word. All objects have to be aligned to 8-byte, otherwise the shadow memory of different objects might overlap. On x86_64 objects on the stack must be 8-byte aligned already and LLWIT does not introduce further overhead. We ensure this alignment in our instrumentation pass for all global and stack allocated data. For heap allocated data our instrumented allocator functions ensure the alignment of allocated memory blocks. The shadow memory setup works only when the number of colors is smaller than 255. The color 0 is reserved for target-specific data, that is not present in the LLVM intermediate code, such as return addresses or saved registers. To guarantee that we do not use more than 255 colors, we randomly merge colors until the number of colors is smaller than 255. Another way to handle larger numbers of colors would be to store 2 or more bytes of color information per 8-byte word. Akritidis et al. [Akr+08] reported that they did not encounter more than 255 colors in their tests. We also did not encounter larger color sizes during our tests on the cyber grand challenge (CGC) dataset, even though we did not implement write safety analysis. During program start, we map the shadow memory into the address space. We rely on the OS to initialize the shadow memory with null bytes. Furthermore, we require that the OS does not allocate physical pages for all virtual pages. Most of the shadow memory stays unused during the execution of the program, but must be mapped. By lazily allocating physical pages, the actual memory usage induced by the shadow memory stays low.

Listing 4.2 shows an instrumented store instruction in the LLVM intermediate code. The first step is to compute the address of the shadow memory based on the pointer, which is stored in the intermediate value %5. Then the color information is loaded from the shadow memory. The color information loaded from the shadow memory is compared to the color information associated with the store instruction. The latter is encoded into the compare instruction as an intermediate. When the comparison succeeds and the coloring is the same, the execution continues normally. Otherwise, we branch to an error handling code. There we call the `__llwit_chk_fail` function from our support library. This function then prints an error message and calls abort to stop the process.

A problem for LLWIT is the static linking of the allocator functions, as it is the case for binaries in the CGC data set. We cannot instrument the functions that are used by the allocator functions, as they operate on different semantics. For example, there might be a helper function that manages in-place free lists, which necessarily writes to objects that are currently not allocated. This is not a real temporal safety violation, as the allocator functions are allowed to do so. Since the CGC binaries cannot link to any other library, they have to provide a custom implementation of the standard allocator functions. Furthermore, the allocator might use other standard functions. For example, an implementation of calloc might use the memset function to fill the allocated data with null bytes. When calloc calls an instrumented memset, the effects are not predictable. The instrumented memset checks the coloring of the passed object, but calloc passes a pointer to a non-initialized memory object. This has the likely outcome that LLWIT will report errors during normal usage. On the other hand, a manipulated call to the memset function might be part of a memory corruption attack, so it should be instrumented.

To solve this problem we have implemented a heuristic in LLWIT to detect which functions are used by the allocator. Our heuristic analyses the call graph provided by LLVM. For each function that is a child node of one of the standard allocator functions, we check whether this function is called from another non-allocator function. If the function is only called from the allocator functions, we consider it part of the allocator implementation and can skip instrumentation. If the function is called from the allocator and from other parts of the program, we duplicate the function. There is one copy of the function for normal usage, that will be instrumented. The second copy is used solely by the allocator and will not be instrumented. We currently do not handle indirect calls used by the allocator functions, which can result in LLWIT producing invalid binaries.

```
; assign numeric color 3 to cgiDir
@cgiDir = common global [64 x i8], align 16

define void @SetCGIDir(i8* %newDir) #0 {
  ;  ...
  %arrayidx5 = getelementptr inbounds [64 x i8],
                        [64 x i8]* @cgiDir, i64 0,
                        i64 %indvars.iv
  ; 1. load numeric color from shadow memory
  ; 1.1 compute shadow memory address
  %2 = ptrtoint i8* %arrayidx5 to i64
  %3 = lshr i64 %2, 3
  %4 = or i64 %3, 2147450880
  %5 = inttoptr i64 %4 to i8*
  ; 1.2 load color from shadow memory
  %6 = load i8, i8* %5
  ; 2. color check
  %7 = icmp ne i8 %6, 3
  br i1 %7, label %8, label %14

; <label>:8:
  ; 3. abort program
  call void @__llwit_chk_fail()
  unreachable

; <label>:14:
  ; 3. continue normal execution
  store i8 %0, i8* %arrayidx5
  ;  ...
}
```

Listing 4.2: LLWIT instrumented store instruction in LLVM IR.

# 5 Evaluation

We evaluated the concept of write-integrity testing (WIT) based on our LLVM-based WIT prototype (LLWIT). Our evaluation is split into two parts. First we used the publicly available cyber grand challenge (CGC) dataset to evaluate LLWIT. We analyse the composition of the colors and the performance of the points-to analysis in determining a good security policy. We identify cases in which LLWIT prevents memory corruption exploits. We compared LLWIT with two mitigation mechanisms included in LLVM: forward-edge control-flow integrity (CFI) and SafeStack (SST). Based on this comparison we identify particular cases where LLWIT does not mitigate memory corruption attacks. In the second part we use the insights gained from the CGC dataset to identify several code patterns that cannot be sufficiently protected by WIT and undermine the security guarantees provided by WIT implementations. We show that the lack of field sensitivity poses a significant problem. Furthermore, we describe some implementation issues, that a field-sensitive WIT implementation faces.

## 5.1 Cyber Grand Challenge Dataset

In the summer of 2016 the, DARPA sponsored, CGC competition ended. The goal of the competition was to foster research into automatic detection, exploitation, and patching of memory corruption vulnerabilities. The competition was split into two events, the CGC qualifying event (CQE) and the CGC final event (CFE). The CQE was used to qualify the top 10 teams for the final event, which was held in parallel to the DEFCON conference. The CFE was structured like a capture-the-flag hacking competition. The idea was to create autonomous systems, called cyber reasoning system (CRS), that compete in such a contest without human intervention. The winning system then competed in the DEFCON capture-the-flag competition against human teams. Capture-the-flag competitions are a game between rivaling teams. Each team is provided the same set of services at the beginning of the game. The services contain vulnerabilities, which can be abused to retrieve the flags. A flag is simply a unique random string, that proves to the organizers, that a vulnerability was successfully exploited. When a team submits a correct flag, the team is awarded points. Usually, there is also some metric, that rewards teams for patching services and keeping them available.

To make it possible that the competition can be played by an autonomous system, the CGC competitions followed a very strict set of rules. First, the target platform for the vulnerable services was a very limited operating system, called the DECREE OS. The DECREE OS is a modified GNU/Linux system. While DECREE features the usual POSIX compatible Linux environment, it also features a special loader and interface for the CGC binaries, called challenge binary (CB). A CB consists of one or more restricted 32-bit `x86` programs. The DECREE OS exposes only 7 system calls to the CBs: `terminate`, `transmit`, `receive`, `fdwait`, `allocate`, `deallocate`, and `random`. There are no system calls that are related to file system usage. The reason for this is that all CB behave idempotent in respect to their input via existing file descriptors. As a consequence a CB cannot save state between recurring executions. The file descriptors are opened by the CB loader. Furthermore, each CB is completely self-contained, the CBs each contain their own standard library and only the `libcgc` is linked to the CB. If a CB consists of more than one binary, the CB loader sets up file descriptors to allow the binaries of the CB to communicate with each other [DL16].

Each event featured its own set of programs, written exclusively for the competition in C and C++. The programs were made publicly available after the competition. For the CQE the teams only had to find a crashing input and patch the binary. During the CFE the CRS had to automatically generate exploits. The CFE featured two types of exploits. The first type of exploit requires taking control over the instruction pointer and one other general purpose register. The second type of exploit must leak a particular page containing random secret data. Since this page is mapped by the operating system (OS) but is not used by the program, an exploit must usually achieve an arbitrary read. The types of vulnerabilities in the programs and the proof of vulnerabilitys (POVs) mimic real vulnerabilities and attacks.

The CBs are rather small and are statically linked, which makes them a perfect target for static analysis techniques. Furthermore, each CB comes with a set of tests, called POLLs, that verify whether the CB is functional. Each CB also comes with at least one POV, an input that triggers a vulnerability in the program. The CBs are written to compile with the LLVM C and C++ compiler `clang` version 3.4. We use the CGC dataset, consisting of all CBs from the CQE and CFE, for evaluation of mitigation mechanisms. For our evaluation we use a port[1] of the CGC dataset to a normal GNU/Linux OS. The port includes a test-driver for running the CBs against all available tests and POVs. We improved[2] the test driver by improving compatibility with modern GNU/Linux distributions, to produce better reports, also in machine-readable formats and added a mechanism to load and store the current state of the test driver. We added support for compiling and testing with different compilation flags, such that we can selectively enable and disable several mitigation mechanisms.

### 5.1.1 Test Setup

We ran the tests and POVs on a large part of the released CBs. We had to disable several CBs because some did not compile with our LLVM toolchain or are broken operating systems other than DECREE. Especially the programs written in C++ use a lot of pointers and multiple levels of indirection, which makes the points-to analysis take longer than feasible. Generally, because of the small size of the CBs, the points-to analysis is computed quickly. However, we encountered a couple of CBs, where the points-to analysis took too long. We aborted compilation and disabled the CB, if the points-to analysis did not finish in under one hour.

We compiled the CBs in different configurations. In all configurations we used LLVM version 3.9, and compiled the CBs with `clang` or `clang++`. For

[1] `https://github.com/trailofbits/cb-multios`

[2] `https://github.com/f0rki/cb-multios/tree/witeval`

the evaluation we use the following five configurations for running the tests:

- Default: CB compiled with clang and no additional settings
- CFI: CB with forward-edge CFI enabled
- SST: CB with SafeStack enabled
- Skip-WIT: CB with the same compilation pipeline as LLWIT (bitcode linking), but with the instrumentation pass disabled. This is used to verify that POVs work.
- WIT: CB with LLWIT instrumentation enabled

We compiled all CBs from the dataset in the variants above. We keep track of the computed WIT colors during compilation with LLWIT. To verify that the compiled CBs are still functional, we used the POLLs. The vast majority of POLLs in the dataset are auto-generated. The `generate-polls` tool uses a specification of the state machine of a CB to generate random walks through the state machine of a CB. We generated 600 POLLs for each CB. For some of the CBs the tool to generate the POLLs fails and we simply ignore those CBs for our evaluation. We then tested all the compiled variants against all POLLs and also all POVs. We test the LLVM CFI and SST mitigation mechanisms in separate configurations. In reality one would combine the forward-edge CFI with a backward-edge protection, such as SST. We test them separately so that we can identify, which of both mechanisms mitigated a POV.

### 5.1.2 Overview and Statistics

We compiled the CBs in five variants and tested all POLLs and POVs against the variants. Table 5.1 shows an overview of the results. The first unfortunate result is that most of the POVs are broken solely by using a different compiler version and different optimization flags. Most POVs do not work against the binaries produced by the Skip-WIT variant, where we compile with the LLWIT pipeline, but without enabling the instrumentation pass. We do not use these CBs for the evaluation, except for the analysis of the color sizes. Some of the CBs lack working POLLs. Those CBs are omitted from the evaluation, as we cannot verify that the mitigation mechanisms do not break the functionality of the tested CBs. When the mitigation mechanism breaks the functionality of a CB, then the POV might be mitigated, or the CB program aborts early. This case is not detected, and it is not easily possible to check whether the vulnerability is actually mitigated in such a case. We simply ignore all those CBs. Furthermore, some CBs fail to work with WIT enabled, because of the different compilation pipeline, or because they contain a WIT violation in their normal functionality. From the 237 CBs, that compile with LLVM 2.9, 208 have working POLLs. LLWIT instrumentation breaks the POLLs for another 81 CBs. We use the remaining 127 CBs, with working POLLs, for analysis of color sizes and statistics over the CBs. The LLWIT compiler pipeline renders some POVs unusable. Furthermore, CFI and SST also break POLLs for 20 CBs. In total this reduces the number of usable CBs to 58, which we use for comparison between LLWIT, CFI, and SST. Of the 58 CBs, LLWIT protects 33 against all POVs.

Table 5.1: Overview of the CGC Evaluation

| CBs with | CB count |
|---|---|
| Available CBs | 248 |
| No Compile Errors (Default) | 237 |
| Working POLLs (Default) | 208 |
| Working POLLs (WIT) | 127 |
| Working POVs (Skip-WIT) and POLLs (WIT, CFI, SST) | 58 |
| All POVs mitigated by WIT | 33 |

We continued to analyse properties of the CBs. First, we analysed the code size of the CBs with successful functional tests. We measured the number of functions and instructions, that are instrumented by LLWIT. We can see in Figure 5.1 the number of instrumented functions and the number of instrumented store instructions for the CBs. Most of the CBs have a rather small code size, in the range of 10 to 20 functions and up to 100 different store instructions. The CGC dataset does not offer a good indication, whether an analysis technique scales to large programs.

We analysed the number of colors for each of the relevant CBs. If a program has a higher color count, the instrumentation of LLWIT can distinguish between more equivalence classes of objects. Figure 5.2 shows the sizes of

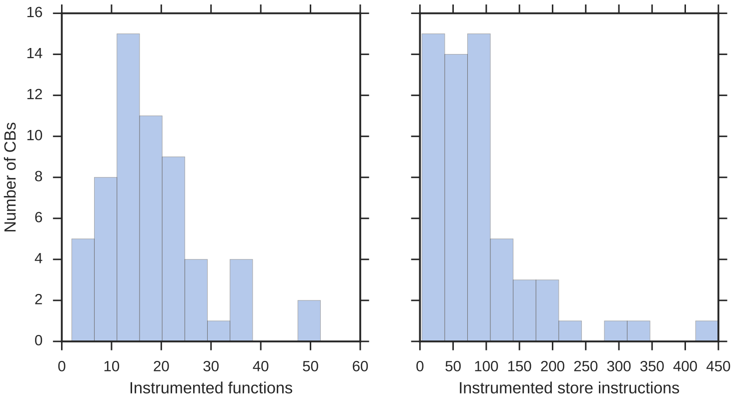


Figure 5.1: Code size of the CBs measured in the number of instrumented functions and store instructions.

the largest color per CB. We can see that the vast majority of CBs have a rather small largest color size. The average size of the largest color is seven objects and 22 store instructions. Still many CBs feature colors that contain both a lot of store instructions and many objects. Of the 245 individual binaries that are part of the CBs, 115 have a color that contains at least 10 store instructions and at least 10 different objects. Note that Figure 5.2 does not include one outlier regarding color size. One color in the FUN CB consists of 18 store instructions and 11742 objects. The vast majority of these objects are constant strings. Any write to those objects would be caught by hardware-enforced read-only memory protection.

An attacker should have fewer opportunities for memory corruption since it is less likely that two memory locations are in the same color. Figure 5.3 shows the number of separate colors per CBs. For CBs consisting of multiple binaries we add the number of colors of each binary, e.g. a CB with two binaries with 2 different colors each would count as a CB with 4 colors. We distinguish between CBs with mitigated and unmitigated POVs. There is no significant difference between CBs with mitigated and unmitigated POVs regarding the size of colors. This suggests that color count alone, can not be used as a measure of security. The type of vulnerability seems to play a more important role for WIT.

A problem for LLWIT is when the points-to analysis returns the universal

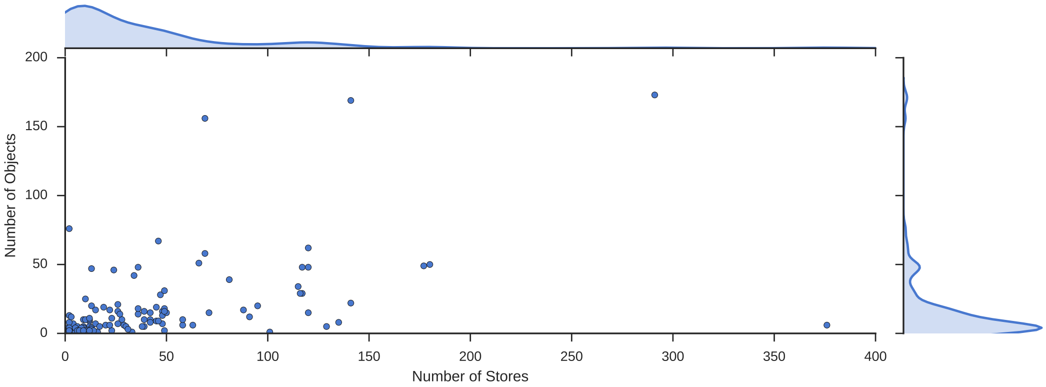


Figure 5.2: Number of Objects and Stores of the largest color per CB.

node. This node represents all possible objects and is returned when the points-to analysis fails to determine a more restricted set. In this case we simply skip instrumentation, since this would collapse all colors into one. On the other hand, not instrumenting a store instruction allows it to write also to control-data. All but four CBs have some store instruction that LLWIT does not instrument. The average number of non-instrumented store instructions is 16.33 with a standard deviation of 29.32. The median is 13 non-instrumented store instructions. We can see that the points-to analysis fails very often and almost all CBs would produce the degenerate case of two colors when instrumented with LLWIT. We omit instrumentation of these instructions with LLWIT to avoid this case. In turn, this also crates holes in the security guarantees that are enforced by LLWIT. All instructions that are not instrumented can be used by an attacker, without any restrictions.

However, we identified dead code, functions that are never actually called by the CB, as one of the reasons for the failure of the points-to analysis. 36 CBs have non-instrumented store instructions inside of a memory or string related helper function, such as `memcpy` or `strcpy`. Since these are often the source for buffer overflow, this would be a worrying result. However, we found that only 12 of the 36 CBs use one of those functions, the other are non-instrumented because they are dead code. Another 24 CBs that have non-instrumented stores within a function, that is related to formatted printing, such as `printf`. Interestingly, not one of the CBs calls any of those functions. The reason for this pattern is, that one group of authors of a

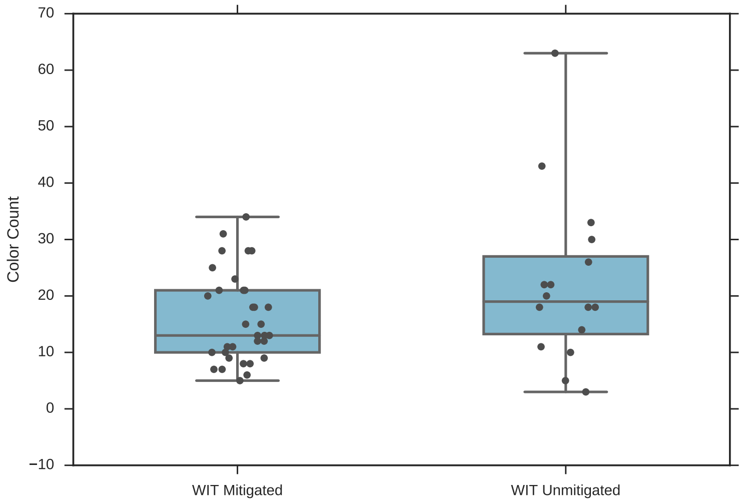


Figure 5.3: Color count for CBs with mitigated and unmitigated POVs.

subset of the CBs reused the same code for libc functions in all their CBs. This results in the same non-instrumented code in many of their CBs.

### 5.1.3 Analysis of Broken CBs

We analysed some of the CBs with broken POLLs. We focused on the CBs, which have interesting properties, such as large colors. We used this to identify weak points and bugs in LLWIT. We discovered several problems with the CGC dataset and also caveats of LLWIT.

**CableGrindLlama** is one of the programs, with broken functionality when LLWIT is enabled. The CB implements a packet dissector and heavily uses function pointers. It uses a function pointer to switch between different `malloc` implementations. Our heuristic cannot deal with this and therefore mistakingly instruments the actual `malloc` implementations. CableGrindLlama is also one of the CBs that take a very long time for the points-to analysis to finish. LLWIT reports 18 different colors, with the largest colors having 757 store instructions and 2603 objects. The reason for this huge color size is that LLWIT instruments the actual `malloc` implementation, which forces a collapse of all heap allocated objects into one color. Without enabling mitigation mechanisms, this CB only passes a few POLLs. With LLWIT enabled even fewer POLLs work.

**Azurad** is a CB written in C++, that implements an interpreter for a simple scripting language. The script is embedded within the CB as a constant string. While the analysis of LLWIT separates the program into 66 different colors, there is one color with 1542 store instructions and 656 different objects. This color is mostly consisting of the tightly coupled code that works with the interpreter state. The LLWIT instrumentation breaks the included parser. The vulnerability is an integer overflow that can be abused to access invalid array indices. The POV is supposed to abuse the integer overflow to perform a data-only information leak attack. The provided POV does not work with LLVM version 3.9. LLWIT would not be able to catch the vulnerability. One array contains all objects related to the state of the interpreter, and any write within the array would be legitimate with LLWIT. Since this is a data-only attack against data on the heap, CFI and SST can not mitigate this vulnerability.

**FASTLANE** is a rather small program, that implements a server for a simple text-based protocol, that mimics HTTP. The vulnerability is a buffer overflow on the stack, which can be used to partially overwrite a pointer. This is a vulnerability that is mitigated with SafeStack. The POV should also be mitigated by LLWIT, but the POLLs are broken with LLWIT enabled. FASTLANE does not use the `malloc` function, but allocates memory directly via the `allocate` system call.

Some CBs, like FASTLANE, use the `allocate` system call to allocate pages and some mix the usage of `allocate` and `malloc`. It is not clear what the semantics of a call to `allocate` are in the context of WIT since such an allocate function is not defined in the C standard. The typical usage of `allocate` is to use it to allocate memory, which is managed with finer granularity by `malloc`. On the other hand the allocated memory can also be treated as one large buffer. Because the semantics of `allocate` is not really clear we simply ignore the `allocate` function. As a result LLWIT either breaks the binary, or misses the opportunity to mitigate a vulnerability.

Similar problems arise when real-world programs use the `mmap` system call on POSIX systems or `VirtualAlloc` on Windows.

### 5.1.4 Comparison with CFI and SafeStack

To identify weak spots of LLWIT we also ran the tests against the CBs with LLVM CFI and SafeStack enabled. Not surprisingly LLWIT managed to mitigate certain vulnerabilities that cannot be mitigated by CFI and SafeStack. Table 5.2 show an overview of the mitigated POVs. We can see that LLWIT generally mitigates more POVs, than CFI and SST combined. Table 5.3 shows a list of CBs, where LLWIT clearly mitigates more POVs than the other mitigations mechanisms. We performed manual analysis of some of the CBs from Table 5.3 to identify why LLWIT performed better than CFI and SafeStack. Some of the POVs from the CQE, do not demonstrate code execution capabilities. They let the CB crash while accessing some controlled but arbitrary address, demonstrating the capability to write or read from an arbitrary addresses. These POV shows the capability to launch further attacks. Since LLWIT verifies the integrity of the store operands, such an invalid access can be detected and mitigated. Since the POV only demonstrates a data-only attack, forward-edge CFI cannot mitigate the POV.

Table 5.2: Comparison of LLWIT with CFI and SafeStack

| CBs with | CB count |
|---|---|
| All POVs mitigated by WIT | 33 |
| All POVs mitigated by CFI or SST | 22 |
| More POVs mitigated by WIT than CFI and SST | 19 |
| More POVs mitigated by SST than WIT | 3 |
| More POVs mitigated by CFI than WIT | 5 |

**Multipass** is an example for a CB, that is exploited with a data-driven attack. The program uses a pointer as a "transaction ID". This leaks the address to the attacker. Using an unaligned transaction ID, the attacker can dereference arbitrary pointers. Using a corrupted pointer to write data is caught by WIT.

**FileSys** is one of the larger CBs, with 124 instrumented store instructions in total. It is written in C++. The vulnerability is a use-after-free bug, that can be exploited by overwriting a vtable pointer. LLWIT catches the invalid write to an object, with a different color. The POV crashes the binary by overwriting the vtable pointer, with a controlled value, but not a real pointer to a pointer. The CB crashes before CFI is able to verify the function pointer loaded from a vtable.

Table 5.3: CBs where WIT mitigates more POVs than CFI and Safestack

| | POVs | POVs mitigated by | | |
|---|---|---|---|---|
| CB Name | Working | WIT | CFI | SST |
| Accel | 2 | 2 | 0 | 0 |
| BudgIT | 2 | 2 | 0 | 0 |
| CGC_Board | 3 | 3 | 0 | 0 |
| FablesReport | 5 | 3 | 0 | 0 |
| FileSys | 1 | 1 | 0 | 0 |
| H2oFlowInc | 1 | 1 | 0 | 0 |
| HeartThrob | 4 | 4 | 0 | 0 |
| LulzChat | 2 | 1 | 0 | 0 |
| Multipass | 1 | 1 | 0 | 0 |
| SLUR_reference_implementation | 2 | 1 | 0 | 0 |
| Sad_Face_Template_Engine_SFTE | 2 | 2 | 0 | 0 |
| Simple_Stack_Machine | 3 | 3 | 0 | 0 |
| UTF-late | 1 | 1 | 0 | 0 |
| Vector_Graphics_Format | 2 | 1 | 0 | 0 |
| cotton_swab_arithmetic | 2 | 2 | 0 | 0 |
| greeter | 4 | 4 | 0 | 0 |
| online_job_application2 | 1 | 1 | 0 | 0 |
| reallystream | 1 | 1 | 0 | 0 |
| virtual_pet | 4 | 4 | 0 | 0 |

While LLWIT generally mitigates more POVs than CFI and SST, we identified a few CBs, where LLWIT performed worse than CFI or SST. By analyzing the vulnerabilities we identified practical limitations a WIT implementation faces. Table 5.4 shows the CBs, where LLWIT failed to mitigate the POVs, but either CFI or SST mitigated more POVs.

Table 5.4: CBs where CFI+Safestack mitigate more than WIT

| | Working | POVs mitigated by | | |
|---|---|---|---|---|
| CB Name | POVs | WIT | CFI | SST |
| Diophantine_Password_Wallet | 5 | 4 | 5 | 0 |
| HackMan | 1 | 0 | 1 | 0 |
| Palindrome | 1 | 0 | 0 | 1 |
| RRPN | 2 | 0 | 2 | 0 |
| payroll | 3 | 0 | 3 | 1 |
| simplenote | 1 | 0 | 1 | 1 |

**Palindrome** is simple program, with a very typical and easy to spot vulnerability. The Palindrome program contains a stack-based buffer overflow, when reading user input. WIT does not mitigate this issue because the out-of-bounds access is performed by the kernel, via the `receive` system call. Since the kernel is not instrumented with LLWIT and does not know about the colorings, the kernel can just write out-of-bounds. Forward-edge CFI does not protect return addresses and fails to mitigate stack-based buffer overflows. SST on the other hand mitigates this issue by placing the buffer on the unsafe stack. The buffer overflow cannot touch the return address on the safe stack.

**HackMan** contains a use of an uninitialized function pointer. The CB is constructed in a way, such that during normal usage the function pointer just happens to be `NULL`. The attacker can control the value on the stack, by abusing a previous stack frame to write at the location of the function pointer. WIT does not mitigate this because uninitialized reads are not covered at all. CFI, on the other hand, mitigates this attack, because it enforces restrictions on the function pointer, whether it was initialized or not.

**payroll** contains a buffer overflow within a data structure, which can be used to corrupt a function pointer. Because LLWIT uses a field-insensitive analysis, it is impossible to distinguish between objects inside of a struct. CFI mitigates this attack because it enforces restrictions on the function pointer upon usage.

**RRPN** implements a just-in-time (JIT) compiler for a simple calculator language. The vulnerability allows the stack and the generated code to collide. The code and the stack are buffers within the same struct. Again because of field-insensitivity WIT fails to detect the corruption of the code pages. Assumptions of all tested mitigation mechanisms are broken because the attacker can corrupt legitimate code pages. Usually, this is mitigated with non-writeable code pages and non-executable data pages. Furthermore, the RRPN CB contains 224 store instructions, where the points-to analysis returned the universal node. LLWIT then skips instrumentation of this store instruction. Surprisingly the POVs fail when CFI is enabled. This is likely the case because running with CFI, breaks assumptions of the POVs.

### 5.1.5 Unmitigated Vulnerabilities

At last we analysed some of the CBs and POVs, that were neither mitigated by LLWIT, CFI, nor SafeStack. This way we identified mostly data-only attacks, that were not mitigated. Table 5.5 shows the CBs, with POVs that were not mitigated by any of the tested mitigation mechanisms. We already identified the use of the unhandled `allocate` system call as the reason for several CBs with broken POLLs. In some cases, the CB stays functional, but LLWIT fails to mitigate the provided POVs. We marked the CBs, that use the `allocate` system call directly in the main source code of the CB. The POVs for those CBs might be mitigated if LLWIT would treat `allocate` the same as `malloc`.

**yolodex** contains a buffer overflow within a data structure. The attacker can corrupt forward and backward pointers of an intrusive linked list. This way the attacker can dereference arbitrary pointers and as a result, achieve an arbitrary read and write exploit primitive. Almost all the functionality of the CB falls within one single color when instrumented with LLWIT. The POV does only demonstrate the arbitrary read, dereferencing arbitrary pointers. Using the vulnerability for an arbitrary write can be caught by LLWIT. However, almost all objects are collapsed within one color, so data-only attacks are likely still possible.

**WordCompletion** is a rather simple CB. It consists of 5 different colors and the largest one consists of 7 store instructions and 1 object. This CB does not use `malloc`, but uses the `allocate` system call directly to allocate memory. LLWIT fails to detect an out-of-bounds write, because it does not handle the `allocate` system call.

Table 5.5: CBs where neither WIT, CFI nor SafeStack mitigate all POVs.

| | Working | POVs mitigated by | | | uses |
|---|---|---|---|---|---|
| CB Name | POVs | WIT | CFI | SST | allocate |
| Audio_Visualizer | 1 | 0 | 0 | 0 | no |
| Enslavednode_chat | 1 | 0 | 0 | 0 | no |
| FablesReport | 5 | 3 | 0 | 0 | no |
| HIGHCOO | 1 | 0 | 0 | 0 | yes |
| Loud_Square_Instant_Messaging_Protocol_LSIMP | 1 | 0 | 0 | 0 | no |
| LulzChat | 2 | 1 | 0 | 0 | no |
| Movie_Rental_Service_Redux | 1 | 0 | 0 | 0 | no |
| Packet_Analyzer | 2 | 0 | 0 | 0 | yes |
| Particle_Simulator | 1 | 0 | 0 | 0 | yes |
| QuadtreeConways | 1 | 0 | 0 | 0 | no |
| SLUR_reference_implementation | 2 | 1 | 0 | 0 | no |
| Tennis_Ball_Motion_Calculator | 1 | 0 | 0 | 0 | yes |
| Vector_Graphics_2 | 2 | 0 | 0 | 0 | yes |
| Vector_Graphics_Format | 2 | 1 | 0 | 0 | no |
| WordCompletion | 1 | 0 | 0 | 0 | yes |
| humaninterface | 2 | 0 | 0 | 0 | no |
| online_job_application | 1 | 0 | 0 | 0 | no |
| yolodex | 1 | 0 | 0 | 0 | no |

Based on the analysis with the CGC dataset we identified several weak spots of LLWIT. We conclude that the failures to mitigate the POVs are a result of the following issues:

- Insufficient modelling of memory allocations (e.g. `allocate`)
- Lack of field-sensitivity (e.g. overflows within a `struct`)
- Misuse of the interface to non-instrumented code (e.g. kernel syscalls)
- Uninitialized read vulnerabilities

We did not observe a POV, which was not mitigated by LLWIT because the target of a write was a different object, but with the same color. The POVs included in the CGC dataset are often not very representative because they were written without considering mitigation mechanisms. An attacker that is aware of the restrictions, a mitigation mechanism imposes, might be able to bypass it. We verified that WIT does indeed mitigate a broader class of vulnerabilities than state-of-the-art mitigation mechanisms, such as CFI and SafeStack. We saw that programs implementing complex functionality, such as an interpreter for a programming language or a parser for binary formats, often contain both a high amount of store instructions and different objects in the same color.

### 5.1.6 Defense against DOP

For data-oriented programming (DOP) it is important that enough gadgets exist to achieve Turing-complete computation. We analysed the sizes of the largest color for each CB. If a color is large, it is more likely that all the necessary DOP gadgets exist within one color. A color that is large in both store instructions and objects is especially attractive to an attacker. The average number of stores per color per CB is 7.33 and the average number of objects per color is 4.98. The average is not really a good measure, since an attacker will try to corrupt data in the largest color available. Figure 5.2 shows the largest colors of the analysed CBs. The average size of the largest color is seven objects and 22 store instructions. Given the size of the equivalence classes created by the coloring, it is very likely that data-oriented attacks are still possible. Given the rather small code size of the CBs in the CGC dataset, we expect that larger programs will also contain larger color sizes.

## 5.2 Problematic Code Patterns

The evaluation with the CGC dataset already revealed some problematic code patterns, where WIT fails to protect against memory corruption attacks. In this section we provide a more detailed analysis of the problems revealed by the evaluation. Furthermore, we discuss consequences of problematic patterns and remaining attack vectors.

When instrumented code interfaces with non-instrumented code, the possibility of undetected memory corruption arises. Every binary interfaces with the operating system kernel via the system call interface. The kernel does not know about object boundaries or LLWIT colors. Therefore the kernel will violate such guarantees when a wrong parameter is passed to a system call. All system calls, that change the contents of user-space memory are susceptible to misuse. Example are the `read` or `recv` system calls. Another example for instrumented code that interfaces with non-instrumented code are JIT compilers. When a binary generates code during runtime, mitigation mechanisms cannot protect the generated code. Hardening JIT compilers requires special care and is very specific to the JIT compiler environment [Ath+15; NT14]. This must be considered out-of-scope for general exploit mitigation mechanisms, such as CFI or WIT.

We learned that for implementing WIT in practice, certain trade-offs must be taken. In LLWIT we opted to skip instrumentation of store instructions, where the points-to analysis failed to return a more specific result than the universal node. We discovered that almost all tested programs in the CGC dataset contained at least a couple of non-instrumented store instructions. These non-instrumented store instructions open up possibilities for an attacker.

A whole class of vulnerabilities, that is simply out-of-scope for WIT, are reads from uninitialized memory. In the usual cases a uninitialized read, might stay unnoticed or result in program crashes. In some cases the attacker is able to control the contents of the uninitialized memory, by abusing previous program inputs. Depending on the type of the uninitialized read this can lead directly to an exploitable attack vector or to further memory corruption. WIT can be combined with a system to protect against uninitialized reads, such as SafeInit [MBG17].

We identified the lack of field-sensitivity in the points-to analysis as the most critical weak point of WIT. All current proposals of data-flow integrity (DFI)-like schemes use a field-insensitive points-to analysis [CCH06; Akr+08; Cad+08; BS08]. If no field-sensitivity is enforced then overflows within a data structure cannot be prevented. We show several problematic code patterns that result in WIT being unable to detect memory corruptions.

### 5.2.1 COOP Attacks and WIT

WIT was proposed as a countermeasure against counterfeit object oriented programming (COOP) attacks [Sch+15]. We show that a field-insensitive variant of WIT is not an effective defense against COOP. C++ code is structured in a way that is not favorable for the static analysis performed by WIT. The object oriented programming model in C++ allows the programmer to call methods on object instances. The way this is implemented is that each method is a normal function, that receives a pointer to the object instance as hidden parameter. This pointer is available as the `this` pointer from within the method. Listing 5.1 shows two methods that utilize the `this` pointer. Both methods are used on the same object, so the points-to analysis concludes that the `this` pointer in both methods can point to the same object. In general it is very likely that the result of the points-to analysis is that the `this` pointer can point to any object of the same type.

```
struct SomeClass {
  int member;
  void methodOne();
  int methodTwo(int j);
};
struct SomeSubClass : public SomeClass {
  int methodThree(int i, int j);
};
// thiscall calling convention: this is passed implicitly
void SomeClass::methodOne(SomeClass* this)
  { this->member++; }
void SomeClass::methodTwo(SomeClass* this, int j)
  { return this->member + j; }

```

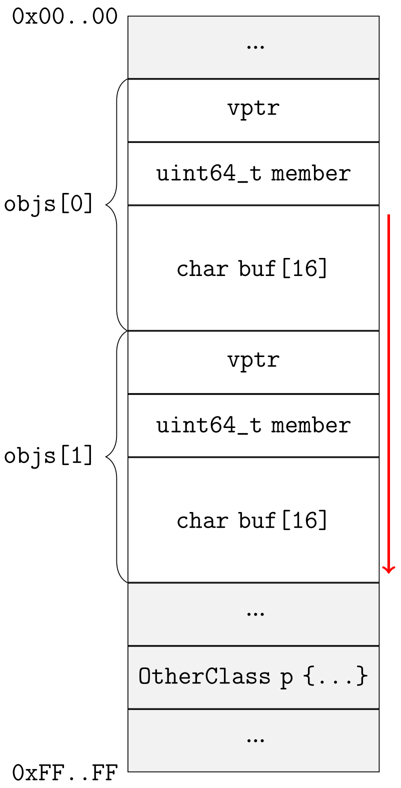
```
int main() {
  SomeClass cls;
  cls.methodOne();
  cls.methodTwo(2);
}
```

Listing 5.1: Illustration of usage of the this pointer in C++.

Because LLWIT and also the original WIT proposal are field-insensitive, they cannot distinguish between the individual members of a class. Our first observation is that all methods of a class, that write to a class member field, will be allowed to write to all the members of all allocation sites of objects of this type. Every write to a member field happens implicitly via the `this` pointer. If the base class has a non-virtual method that writes to a non-static member, then there exists a method that can write to any object that is derived from the base class. Subsequently, WIT is forced to put all objects into the same color. As a result the whole type hierarchy will be collapsed into the same color. An attacker can use any method from any subclass also on a base class without triggering a WIT violation.

The policy enforced by a field-insensitive WIT gives an attacker a lot of possibilities to create an exploit by abusing all methods and all objects associated with one type hierarchy. To launch a COOP attack the attacker must first corrupt a virtual method table (vtable) pointer. The placement of the vtable pointer is implementation specific and can be chosen by the compiler. Most compilers place the vtable pointer at the top of the allocated object. To corrupt the vtable pointer the attack must violate spatial or temporal safety. WIT can catch many such violations. However, one problematic pattern are continuous chunks of memory consisting of different objects, but all of the same color. An attacker could overflow from the first of the objects into the second or further, without violating the WIT policy. To demonstrate how severe this problem is, we give an example in which an attacker can overwrite a `vtable` pointer. Figure 5.4 shows a listing of problematic code on the right. This code then produces a memory layout as shown on the left. The attacker can now overflow from the first object, into the second object and overwrite a `vtable` pointer.

Furthermore such continuous memory chunks must not necessarily be created using a single allocation with `malloc`. For example the `jemalloc`



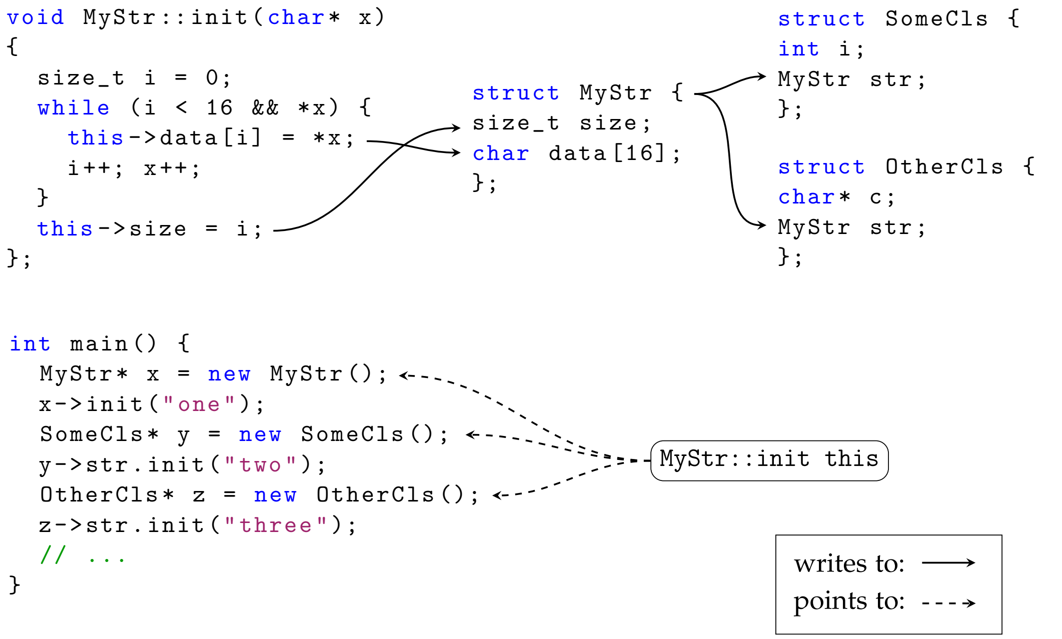
```
class SomeClass {
  uint64_t member;
  // prone to a buffer overflow
  char buf[16];

  virtual ~SomeClass();
  virtual void virtMethod();
}
// ...
vector<SomeClass> objs;
for (size_t i = 0; i < 2; i++)
  objs.push_back(SomeClass());
OtherClass* p = new OtherClass();
// ...
gets(objs[0].buf);
```

Figure 5.4: Example for a problematic pattern for WIT, due to adjacent objects with the same color. WIT would not prevent the vptr overwrite.

heap implementation does not store meta-data in front of allocated chunks, but keeps the meta-data in a separate memory area. This is in contrast to the `ptmalloc`-based allocators, such as the one in the GNU libc. This allows `jemalloc` to put objects right beside each other without any gaps. An attacker can trigger two allocations of the same size and color and get a continuous chunk of memory of one color. The allocator used alongside any DFI implementation should insert red-zones in between objects, such that it can catch out of bounds accesses.

A possible solution to this was proposed for data space randomization [BS08]. One can leverage the concept of pool allocation [LA05], but instead of allocating objects of the same type in the same pool, objects of the same type and color are allocated in different pools. The pools are then separated with guard pages. This ensures that objects with the same color are never allocated in the same memory region and are safe from linear buffer overflows. The same problem with arrays cannot be as easily solved. An array type is defined to be a continuous chunk of memory, by the C/C++ language standards. The instrumentation code would have to break this definition and allocate each array member in a different pool and instrument all array access operations to redirect to the actual memory location of the array field. This would break assumptions about the memory layout of objects and cause incompatibilities with existing code. Furthermore it would cause a severe performance degradation by breaking data caching of the CPU.

We identified another problematic pattern: compound data structures. This problem is not C++ specific, but is also a common pattern in C code. When data structures are nested then WIT must collapse both types into the same color, as can be seen in the example in Figure 5.5. Because `MyStr` is embedded in `SomeCls`, the `init` method of `MyStr` must be allowed to access data that is stored within `SomeCls` objects. Because of the field insensitivity, `init` is also allowed to write to any other member of `SomeCls`. Additionally, if the embedded class `MyStr` is also embedded in a totally unrelated class `OtherCls`, this class will also be put into the same color as `MyStr` and therefore also `SomeCls`. The points-to node of the `this` pointer in the `init` method, contains all three objects `x, y, z` of different types. Under WIT both store instructions in the `init` method can therefore write to any member of all given classes.



```
void MyStr::init(char* x)
{
  size_t i = 0;
  while (i < 16 && *x) {
    this->data[i] = *x;
    i++; x++;
  }
  this->size = i;
};
```

```
struct MyStr {
size_t size;
char data[16];
};
```

```
struct SomeCls {
int i;
MyStr str;
};

struct OtherCls {
char* c;
MyStr str;
};
```

```
int main() {
  MyStr* x = new MyStr();
  x->init("one");
  SomeCls* y = new SomeCls();
  y->str.init("two");
  OtherCls* z = new OtherCls();
  z->str.init("three");
  // ...
}
```

Figure 5.5: A field-insensitive points-to analysis collapses all fields together, resulting in impossible points-to nodes.

Summarizing, WIT does not fully prevent COOP attacks. However, WIT limits an attacker in her capabilities to construct a COOP attack. The attacker must operate within the static data-flow graph for all write operations. We showed, that common C++ code patterns allow an attacker a high degree of freedom within the enforced security policy of a field-insensitive WIT. While WIT restricts an attacker from reusing virtual methods from a different WIT color, C++ makes it very likely that many methods will fall into the same color. All objects within one type hierarchy and all embedded data structures will stay within one color. It is then also more likely that an attacker has all necessary COOP gadgets available. In summary we showed that WIT alone does not offer comprehensive protection against COOP. It is therefore crucial that WIT is combined with a precise CFI solution as in the original WIT proposal [Akr+08] or some other specialized vtable protection method [PHY15].

### 5.2.2 Problems with Field Sensitivity

All previous proposals of DFI schemes, such as DFI, WIT, data randomization and data-space randomization, used a field-insensitive points-to analysis and noted that using a field-sensitive points-to analysis is future work [CCH06; Akr+08; BS08; Cad+08]. Indeed a field-sensitive points-to analysis would allow WIT to prevent the attacks described previously in this section. We argue that implementing a field-sensitive version of WIT faces many compatibility issues and might not be suitable to harden legacy codebases.

A very nice property about WIT, and all other DFI schemes, is that they do not suffer from false positives. If WIT detects an error, then the program has a bug. This makes WIT practical to secure legacy code, by recompiling it. Techniques that require changes or even annotations to the source code are mostly not practical for this purposes. Introducing field-sensitivity creates issues that break compatibility with existing software. For example, the C/C++ languages guarantee that a simple `struct` or a "trivially copyable type", for example a C++ plain old data type (POD), can be copied from one memory location to another with `memcpy`. An object can also be copied into an array of type `unsigned char` as a temporary storage. The language guarantees that these copy operations will result in exact copies of these objects [ISO12; ISO14]. This poses a problem for a field-sensitive implementations of WIT. To achieve compatibility with the language, the `memcpy` function would need to be treated in a special way, as `memcpy` would be allowed to copy objects in bulk ignoring the field specific colorings. A simple solution would be to allow `memcpy` to completely ignore the coloring. However, this opens up opportunities for an attacker, as many string handling functions rely on calling `memcpy`. A more sophisticated solution would introduce a special version of `memcpy` for copying objects. It is questionable whether this case can always be detected with static analysis. It would actually be permissible to implement a custom `memcpy` implementation, which would be infeasible for static analysis to detect. Furthermore in a field-sensitive WIT implementation, the data structures would become much bigger. Every field would have to be aligned to 8 bytes. This would increase memory usage, much more than WIT does anyway.

### 5.2.3 Attacks Against Allocators

DFI, WIT and similar schemes usually do not explicitly consider temporal safety. While it is definitely a bug, it is not clear, whether data-flow to or from deallocated objects is considered as invalid by DFI schemes. The object is still the same, although it is technically not existing anymore. Only when a different object, with a different color, is allocated at the same memory location, a data-flow integrity violation is clearly happening. In our WIT prototype we color freed objects with the default color 0, if we can retrieve the size of the freed object. This way write-after-free bugs can be detected. Other temporal safety violations cannot be detected by WIT. For example if a different object, but with the same color, is allocated in the same memory location, then a temporal safety violation would stay undetected.

Furthermore WIT, and also all other DFI schemes, cannot instrument the allocator itself. The allocator implementation cannot operate on the same set of objects as a normal C program, but must manage memory at a different level. For example the GNU libc allocator writes to freed memory, to manage in-place free lists. Subsequently DFI schemes must assume that the allocator is correct.

Several memory corruption attacks against memory allocators have been studied [Pha05; bla09; ah12]. These attacks abuse the internals of the allocator's implementation. Some of these attacks are mitigated by WIT. For example the traditional unlink technique requires a corruption of a freed chunk to modify the forward and backward pointers of the allocator free list. This write to a freed object would be detected by LLWIT. Other attacks are impossible to detect. For example double free vulnerabilities can cause an inconsistent state in the allocators internal meta data. Listing 5.2 shows a snippet of code that illustrates how a double free can make the allocator return the same address twice. The `glibc` allocator features only rudimentary double free detection, that detect only two consecutive calls to free with the same address.

```
// double free in ptmalloc2 based glibc allocator
char* a = malloc(8);
char* b = malloc(8);
char* c = malloc(8);

free(a);
free(b); // avoid double free detection
free(a);
// free list: [a, b, a]
char* x = malloc(8);
// free list: [b, a]
char* y = malloc(8);
// free list: [a]
char* z = malloc(8);
// the following holds
assert(x == z);
```

Listing 5.2: Double free vulnerability.

Such attacks cannot be detected by WIT alone. An attacker can use such a bug to create a condition, where two pointers to logically different objects point to the same memory location. This scenario is similar to use-after-free bugs. With LLWIT enabled the attacker is restricted to only use the last allocated pointer for writing, since the color of the shadow memory is determined by the last object allocated at that address. The attacker can then use the new object to write to the memory location and use the older object for reading. If both objects happen to have the same coloring, then the attacker is not restricted at all.

Listing 5.3 shows a snippet of code that would allow an attacker to perform such an attack. The `login` function contains a double free bug, similar to the Listing 5.2. By triggering the `login` function a second time, the attacker can make the `creds` pointer point to the same memory location as `otp_user_token`. The attacker can then write to the `otp_user_token` on line 14. The attacker now freely controls the credential structure behind the `creds` pointer and can set the `uid` field to zero, to pass the check on line 16. This is a purely data-oriented attack. But a similar attack can be used to corrupt function pointers or vtable pointers in C++ objects.

Although this attack is a particular weak spot of WIT, other attacks against allocators usually require corruption of internal meta-data. The majority of attack vectors for corrupting heap meta-data is mitigated by WIT, e.g. writing to a freed chunk can be prevented by WIT. Attacks against allocators are an example of attacks against non-instrumented parts of the program.

```
// invoked 2 times in a row by the attacker
void login(uint64_t uid) {
  // 1st malloc
  creds = malloc(sizeof(creds_t) /* == OTP_SIZE */);
  creds->uid = uid;
  // 2nd malloc
  another_allocation = malloc(OTP_SIZE);
  // 3rd malloc - will contain attacker input
  otp_user_token = malloc(OTP_SIZE);

  if (creds->uid == 0) {
    puts("root login disabled");
  } else {
    if (verify_token(uid,
                     // reads malicious input from user
                     otp_user_token)) {
      if (creds->uid == 0) {
        // SUCCESS: root shell
        // ...
      } else {
        // ...
  }}}
  free(creds);
  free(otp_user_token);
  // BUG: double free
  free(creds);
  // freelist: [creds, otp_user_token, creds]
}
```

Listing 5.3: Code that allows an attacker to bypass LLWIT.

# 6 Conclusion

In this thesis, we evaluated the concept of write-integrity testing (WIT). WIT is an exploit mitigation scheme that enforces a subset of data-flow integrity (DFI) and promises protection against data-only attacks. We showed that WIT does indeed protect against control-flow hijacking and data-only attacks. Our LLVM-based WIT prototype (LLWIT) was able to mitigate all vulnerabilities in 33 of 58 programs. However, we also identified various code patterns that lead to weak security guarantees offered by WIT. We showed that a field-insensitive WIT does not offer comprehensive protection against counterfeit object oriented programming (COOP).

We implemented WIT within the LLVM compiler framework because no WIT implementation is publicly available. We described the challenges of implementing WIT. When applying WIT, we opted to use static linking and whole-program analysis, so that the points-to analysis can track the points-to information through library functions. We had to implement a heuristic to separate the allocator implementation from the rest of the code during whole-program analysis. Based on LLWIT, we performed an analysis on the security guarantees offered by WIT implementations.

In the first part of our evaluation, we used a set of programs from the cyber grand challenge (CGC) competition. We compiled all programs with LLWIT instrumentation enabled, and tested them with the provided functional tests and the provided proof of vulnerability (POV) exploits. Our first lesson with the CGC dataset was that most POVs are not functional when a different compiler version is used. We used the remaining programs in the CGC dataset to evaluate WIT and compare it to state-of-the-art mitigation mechanisms: control-flow integrity (CFI) and SafeStack. As expected, LLWIT generally prevents exploitation of more POVs than CFI and SafeStack. Our implementation mitigates all vulnerabilities in 33 of 58 programs out of the CGC dataset. CFI and SafeStack combined, mitigate more vulnerabilities than LLWIT in 6 programs. We saw that WIT is not able to mitigate uninitialized read vulnerabilities, protect programs that use a just-in-time (JIT)-compiler, and can have issues when WIT-instrumented code interfaces with non-instrumented code. The weakest point of WIT lies in the usage of a field-insensitive points-to analysis.

In the second part of our evaluation, we used the insights gained from the CGC dataset to identify problematic code patterns. Common code patterns in C++ programs decrease the security guarantees that are offered by WIT. We demonstrated that an attack, that abuses double free bugs, can allow an attacker to violate temporal safety and bypass WIT. WIT should, therefore, be combined with a hardened allocator, that offers better double-free detection than the GNU libc heap implementation. We showed that WIT cannot offer sufficient protection against COOP attacks. The main problem is again the lack of field-sensitivity. We showed code patterns that allow an attacker to corrupt the virtual method table (vtable) pointer of an object, even when protected with WIT. Furthermore, WIT does not restrict an attacker enough, so that she can still construct useful COOP attacks. Therefore, it is essential that WIT is also combined with a CFI solution, which is aware of C++ semantics. However, in big C++ code bases with a lot of inheritance, an attacker might be able to bypass both WIT and CFI.

We conclude that WIT alone does not offer comprehensive protection against memory corruption attacks in general. Many of the problems of WIT are also applicable to other DFI schemes. The problem of mitigating data-only memory corruption attacks with practical overhead is still an open research topic.

## Future Work

We already identified various code patterns that lead to a weak policy, enforced by WIT. However, we only analyzed rather small programs in the CGC dataset. Further analysis on how WIT performs on large code bases, such as a web browser, is needed. Our results suggest that WIT would perform even worse when the code base is large. To work with large codebases, a faster points-to analysis is needed. Using an Andersen-style points-to analysis results in a points-to graph with high precision at the cost of cubic runtime. This precision is not needed because WIT merges points-to nodes in the color sets. A Steensgard-style points-to analysis would allow WIT to analyze large code bases in nearly linear time.

We identified field-insensitivity as the main problem of WIT. Other DFI schemes suffer from similar weaknesses. While a field-insensitive WIT is highly compatible with existing non-instrumented code, a field-sensitive version of WIT would have to break certain assumptions and guarantees of the C/C++ languages. It would be interesting to verify if a field-sensitive version of WIT could be implemented to work on real-life programs.

# Appendix

# Acronyms

**ASLR** adress space layout randomization
**BROP** blind return oriented programming
**CB** challenge binary
**CFE** CGC final event
**CFG** control-flow graph
**CFI** control-flow integrity
**CGC** cyber grand challenge
**CGI** common gateway interface
**COOP** counterfeit object oriented programming
**CPI** code pointer integrity
**CPU** central processing unit
**CQE** CGC qualifying event
**CRS** cyber reasoning system
**DEP** data execution prevention
**DFI** data-flow integrity
**DOP** data-oriented programming
**DoS** denial of service
**GOT** global offset table
**JIT** just-in-time
**JIT-ROP** just-in-time return oriented programming
**LLWIT** our LLVM-based WIT prototype
**MAC** message authentication code
**OS** operating system
**POD** plain old data type
**POV** proof of vulnerability
**RDT** runtime definitions table
**RELRO** read-only relocations
**ret2libc** return to lib(c)
**ROP** return-oriented programming
**SFI** software fault isolation
**SSA** single static assignment
**SST** SafeStack
**URL** uniform resource locator
**vtable** virtual method table
**WIT** write-integrity testing

# Bibliography

[14] *The Hearbleed Bug*. 2014. URL: http://heartbleed.com/ (visited on 01/18/2017) (cit. on p. 24).

[Aba+05] Martín Abadi et al. “Control-flow integrity.” In: *Proceedings of the 12th ACM conference on Computer and communications security*. ACM. 2005, pp. 340–353 (cit. on pp. 5, 33, 35, 37).

[Aba+09] Martín Abadi et al. “Control-flow integrity principles, implementations, and applications.” In: *ACM Transactions on Information and System Security (TISSEC)* 13.1 (2009), p. 4 (cit. on p. 35).

[ah12] argp and huku. *Pseudomonarchia jemallocum*. 2012. URL: http://phrack.org/issues/68/10.html (visited on 04/07/2017) (cit. on p. 77).

[Akr+08] Periklis Akritidis et al. “Preventing memory error exploits with WIT.” In: *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. IEEE. 2008, pp. 263–277 (cit. on pp. 6, 20, 21, 41, 45, 48, 51, 70, 75).

[ASU86] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, Principles, Techniques*. Addison wesley Boston, 1986 (cit. on p. 41).

[Ath+15] Michalis Athanasakis et al. “The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines.” In: *NDSS*. 2015 (cit. on p. 69).

[Bac+14] Michael Backes et al. “You can run but you can’t read: Preventing disclosure exploits in executable code.” In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2014, pp. 1342–1353 (cit. on p. 30).

[BB14] Erik Bosman and Herbert Bos. "Framing signals-a return to portable shellcode." In: *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE. 2014, pp. 243–258 (cit. on p. 19).

[Bit+14] Andrea Bittau et al. "Hacking blind." In: *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE. 2014, pp. 227–242 (cit. on pp. 19, 29).

[bla09] blackngel. *MALLOC DES-MALEFICARUM*. 2009. URL: `http://phrack.org/issues/66/10.html` (visited on 04/07/2017) (cit. on p. 77).

[Ble+11] Tyler Bletsch et al. "Jump-oriented programming: a new class of code-reuse attack." In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM. 2011, pp. 30–40 (cit. on p. 19).

[BN14] Michael Backes and Stefan Nürnberger. "Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing." In: *USENIX Security*. Vol. 14. 2014 (cit. on p. 31).

[BS08] Sandeep Bhatkar and R Sekar. "Data space randomization." In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2008, pp. 1–22 (cit. on pp. 4, 41, 44, 70, 73, 75).

[Cad+08] Cristian Cadar et al. *Data randomization*. Tech. rep. Technical Report TR-2008-120, Microsoft Research, 2008. Cited on, 2008 (cit. on pp. 41, 44, 70, 75).

[Car+15] Nicolas Carlini et al. "Control-flow bending: On the effectiveness of control-flow integrity." In: *24th USENIX Security Symposium, USENIX Sec*. 2015 (cit. on pp. 5, 37, 39).

[CCH06] Miguel Castro, Manuel Costa, and Tim Harris. "Securing software by enforcing data-flow integrity." In: *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association. 2006, pp. 147–160 (cit. on pp. 6, 41, 42, 44, 70, 75).

[Che+05] Shuo Chen et al. "Non-Control-Data Attacks Are Realistic Threats." In: *Usenix Security*. Vol. 5. 2005 (cit. on p. 20).

[Che+10] Stephen Checkoway et al. "Return-oriented programming without returns." In: *Proceedings of the 17th ACM conference on Computer and communications security*. ACM. 2010, pp. 559–572 (cit. on p. 19).

[Con+15] Mauro Conti et al. "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*. ACM. 2015, pp. 952–963 (cit. on p. 37).

[Cow+03] Crispin Cowan et al. "Pointguard TM: protecting pointers from buffer overflow vulnerabilities." In: *Proceedings of the 12th conference on USENIX Security Symposium*. Vol. 12. 2003, pp. 91–104 (cit. on p. 32).

[Cow+99] Crispin Cowan et al. "Protecting systems from stack smashing attacks with StackGuard." In: *Linux Expo*. 1999 (cit. on pp. 5, 39).

[Cra+15a] Stephen J Crane et al. "It's a TRaP: Table randomization and protection against function-reuse attacks." In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2015, pp. 243–255 (cit. on pp. 19, 32).

[Cra+15b] Stephen Crane et al. "Readactor: Practical code randomization resilient to memory disclosure." In: *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE. 2015, pp. 763–780 (cit. on p. 32).

[DA06] Dinakar Dhurjati and Vikram Adve. *Backwards-Compatible Array Bounds Checking for C with Very Low Overhead*. Tech. rep. Shanghai, China, 2006. URL: `http://llvm.org/pubs/2006-05-24-SAFECode-BoundsCheck.html` (cit. on p. 26).

[Dav+12a] Lucas Davi et al. "MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones." In: *NDSS*. Vol. 2. 6. 2012, p. 27 (cit. on p. 37).

[Dav+12b] Lucas Davi et al. "XIFER: a software diversity tool against code-reuse attacks." In: *4th ACM International Workshop on Wireless of the Students, by the Students, for the Students (S3 2012)*. Vol. 174. 2012 (cit. on p. 30).

[Dav+14] Lucas Davi et al. “Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection.” In: *USENIX Security Symposium*. 2014 (cit. on pp. 35, 37).

[Dav+15] Lucas Davi et al. “Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming.” In: *NDSS*. 2015 (cit. on p. 32).

[Dhu+03] Dinakar Dhurjati et al. “Memory safety without runtime checks or garbage collection.” In: *ACM SIGPLAN Notices* 38.7 (2003), pp. 69–80 (cit. on p. 26).

[DKA06] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. “SAFECode: enforcing alias analysis for weakly typed languages.” In: *PLDI ’06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*. Ottawa, Ontario, Canada: ACM, 2006, pp. 144–157. ISBN: 1-59593-320-4. DOI: `http://doi.acm.org/10.1145/1133981.1133999` (cit. on p. 26).

[DL16] DARPA and LegitBS. *DARPA Cyber Grand Challenge documentation*. 2016. URL: `https://cgc-docs.legitbs.net/` (visited on 04/21/2017) (cit. on p. 55).

[Eva+15] Isaac Evans et al. “Missing the Point (er): On the Effectiveness of Code Pointer Integrity1.” In: *IEEE Symp. on Security and Privacy*. 2015 (cit. on pp. 5, 23, 40).

[FBI15] FBI. *Ransomware on the Rise*. 2015. URL: `https://www.fbi.gov/news/stories/ransomware-on-the-rise` (visited on 01/15/2017) (cit. on p. 1).

[Gaw+16a] Robert Gawlik et al. “Detile: Fine-grained information leak detection in script engines.” In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 322–342 (cit. on p. 30).

[Gaw+16b] Robert Gawlik et al. “Enabling client-side crash-resistance to overcome diversification and information hiding.” In: *Symposium on Network and Distributed System Security (NDSS)*. 2016 (cit. on pp. 23, 29).

[Gib] Samuel Gibbs. *Dropbox hack leads to leaking of 68m user passwords on the internet*. URL: `https://www.theguardian.com/technology/2016/aug/31/dropbox-hack-passwords-68m-data-breach` (visited on 01/15/2017) (cit. on p. 1).

[Gok+14] Enes Goktas et al. "Out of control: Overcoming control-flow integrity." In: *Security and Privacy (SP), 2014 IEEE Symposium on*. IEEE. 2014, pp. 575–589 (cit. on p. 37).

[Gro+02] Dan Grossman et al. "Region-based memory management in Cyclone." In: *ACM Sigplan Notices* 37.5 (2002), pp. 282–293 (cit. on p. 27).

[Gua16] The Guardian. *Hackers jailed over SpyEye virus that robbed bank accounts worldwide*. 2016. URL: `https://www.theguardian.com/technology/2016/apr/21/hackers-jailed-over-spyeye-virus-that-robbed-bank-accounts-worldwide` (visited on 01/15/2017) (cit. on p. 1).

[Her16] Alex Hern. *Ransomware threat on the rise as 'almost 40% of businesses attacke'*. 2016. URL: `https://www.theguardian.com/technology/2016/aug/03/ransomware-threat-on-the-rise-as-40-of-businesses-attacked` (visited on 01/15/2017) (cit. on p. 1).

[Hic+04] Michael Hicks et al. "Experience with safe manual memory-management in cyclone." In: *Proceedings of the 4th international symposium on Memory management*. ACM. 2004, pp. 73–84 (cit. on p. 27).

[Hu+16] Hong Hu et al. "Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks." In: (2016) (cit. on pp. 5, 6, 21).

[ISO12] ISO ISO. "ISO/IEC 9899-2011: Information technology – Programming languages – C." In: *ISO Working Group* (2012) (cit. on p. 76).

[ISO14] ISO ISO. "ISO/IEC 14882-2014: Information technology – Programming languages – C++." In: *ISO Working Group* (Dec. 2014) (cit. on p. 76).

[Kil+06] Chongkyung Kil et al. "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software." In: *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE. 2006, pp. 339–348 (cit. on pp. 4, 30).

[KKP03] Gaurav S Kc, Angelos D Keromytis, and Vassilis Prevelakis. "Countering code-injection attacks with instruction-set randomization." In: *Proceedings of the 10th ACM conference on Computer and communications security*. ACM. 2003, pp. 272–280 (cit. on p. 32).

[Kra05] Sebastian Krahmer. *x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique*. 2005 (cit. on p. 17).

[Kus13] David Kushner. *The Real Story of Stuxnet*. 2013. URL: `http://spectrum.ieee.org/telecom/security/the-real-story-of-stuxnet/` (visited on 01/15/2017) (cit. on p. 2).

[Kuz+14] Volodymyr Kuznetsov et al. "Code-pointer integrity." In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2014 (cit. on pp. 5, 7, 40).

[LA04] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California, 2004 (cit. on pp. 6, 48).

[LA05] Chris Lattner and Vikram Adve. "Automatic pool allocation: improving performance by controlling data structure layout in the heap." In: *ACM SIGPLAN Notices* 40.6 (2005), pp. 129–142 (cit. on pp. 26, 73).

[Mas+09] Joshua Mason et al. "English shellcode." In: *Proceedings of the 16th ACM conference on Computer and communications security*. ACM. 2009, pp. 524–533 (cit. on p. 16).

[Mas+15] Ali José Mashtizadeh et al. "CCFI: Cryptographically enforced control flow integrity." In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2015, pp. 941–951 (cit. on p. 39).

[MBG17] Alyssa Milburn, Herbert Bos, and Cristiano Giuffrida. "SafeInit: Comprehensive and Practical Mitigation of Uninitialized Read Vulnerabilities." In: (2017) (cit. on p. 69).

[Nag+09] Santosh Nagarakatte et al. "SoftBound: highly compatible and complete spatial memory safety for c." In: *ACM Sigplan Notices*. Vol. 44. 6. ACM. 2009, pp. 245–258 (cit. on p. 27).

[Nag+10] Santosh Nagarakatte et al. "CETS: compiler enforced temporal safety for C." In: *ACM Sigplan Notices*. Vol. 45. 8. ACM. 2010, pp. 31–40 (cit. on p. 27).

[Nag12] Santosh Ganapati Nagarakatte. "Practical low-overhead enforcement of memory safety for c programs." PhD thesis. University of Massachusetts Amherst, 2012 (cit. on pp. 4, 26).

[NMW02] George C Necula, Scott McPeak, and Westley Weimer. "CCured: Type-safe retrofitting of legacy code." In: *ACM SIGPLAN Notices*. Vol. 37. 1. ACM. 2002, pp. 128–139 (cit. on p. 27).

[NS07] Nicholas Nethercote and Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation." In: *ACM Sigplan notices*. Vol. 42. 6. ACM. 2007, pp. 89–100 (cit. on p. 3).

[NT14] Ben Niu and Gang Tan. "RockJIT: Securing just-in-time compilation using modular control-flow integrity." In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2014, pp. 1317–1328 (cit. on p. 69).

[One96] Aleph One. *Smashing The Stack For Fun And Profit*. 1996. URL: `http://phrack.org/issues/49/14.html` (visited on 12/20/2016) (cit. on pp. 5, 10, 25).

[Pap12] Vasilis Pappas. "kBouncer: Efficient and transparent ROP mitigation." In: *tech. rep. Citeseer* (2012) (cit. on p. 35).

[PBG15] Mathias Payer, Antonio Barresi, and Thomas R Gross. "Fine-grained control-flow integrity through binary hardening." In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2015, pp. 144–164 (cit. on p. 37).

[Pha05] Phantasmal Phantasmagoria. *The Malloc Maleficarum - Glibc Malloc Exploitation Techniques*. 2005. URL: `http://seclists.org/bugtraq/2005/Oct/118` (visited on 04/07/2017) (cit. on p. 77).

[PHY15] Aravind Prakash, Xunchao Hu, and Heng Yin. "vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries." In: *NDSS*. 2015 (cit. on p. 75).

[Pro16] Emil Protalinski. *Pwn2Own 2016: Chrome, Edge, and Safari hacked, £460,000 awarded in total*. 2016. URL: `http://venturebeat.com/2016/03/18/pwn2own-2016-chrome-edge-and-safari-hacked-460k-awarded-in-total/` (visited on 01/15/2017) (cit. on p. 2).

[Raw+17] Sanjay Rawat et al. "VUzzer: Application-aware Evolutionary Fuzzing." In: (2017) (cit. on p. 4).

[rix01] rix. *Writing ia32 alphanumeric shellcodes*. 2001. URL: `http://phrack.org/issues/57/15.html#article` (visited on 01/25/2017) (cit. on p. 16).

[Rog+] Roman Rogowski et al. "Revisiting Browser Security in the Modern Era: New Data-only Attacks and Defenses." In: () (cit. on p. 23).

[SB13] Matthew S Simpson and Rajeev K Barua. "MemSafe: ensuring the spatial and temporal memory safety of C at runtime." In: *Software: Practice and Experience* 43.1 (2013), pp. 93–128 (cit. on p. 27).

[Sch+14] Felix Schuster et al. "Evaluating the effectiveness of current anti-ROP defenses." In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2014, pp. 88–108 (cit. on p. 35).

[Sch+15] Felix Schuster et al. "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications." In: *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE. 2015, pp. 745–762 (cit. on pp. 6, 7, 19, 70).

[Sch10] Bruce Schneier. *The Story Behind The Stuxnet Virus*. 2010. URL: http://www.forbes.com/2010/10/06/iran-nuclear-computer-technology-security-stuxnet-worm.html (visited on 01/15/2017) (cit. on p. 1).

[Ser+12] Konstantin Serebryany et al. "AddressSanitizer: A Fast Address Sanity Checker." In: *USENIX ATC 2012*. 2012. URL: https://www.usenix.org/conference/usenixfederatedconferencesweek/addresssanitizer-fast-address-sanity-checker (cit. on pp. 3, 51).

[Sha+04] Hovav Shacham et al. "On the effectiveness of address-space randomization." In: *Proceedings of the 11th ACM conference on Computer and communications security*. ACM. 2004, pp. 298–307 (cit. on pp. 23, 29).

[Sha07] Hovav 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*. ACM. 2007, pp. 552–561 (cit. on pp. 5, 17).

[Sha14] Stephen Shankland. *'Heartbleed' bug undoes Web encryption, reveals Yahoo passwords*. 2014. URL: https://www.cnet.com/news/heartbleed-bug-undoes-web-encryption-reveals-user-passwords/ (visited on 01/18/2017) (cit. on p. 24).

[Sno+13] Kevin Z Snow et al. "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization." In: *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE. 2013, pp. 574–588 (cit. on pp. 19, 23).

[Sno+16] Kevin Z Snow et al. "Return to the zombie gadgets: Undermining destructive code reads via code inference attacks." In: *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE. 2016, pp. 954–968 (cit. on p. 31).

[Son+16] Chengyu Song et al. "Enforcing Kernel Security Invariants with Data Flow Integrity." In: (2016) (cit. on p. 21).

[Ste+16] Nick Stephens et al. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution." In: *Proceedings of the Network and Distributed System Security Symposium*. 2016 (cit. on p. 4).

[Str+09] Raoul Strackx et al. "Breaking the memory secrecy assumption." In: *Proceedings of the Second European Workshop on System Security*. ACM. 2009, pp. 1–8 (cit. on p. 29).

[Sze+13] László Szekeres et al. "Sok: Eternal war in memory." In: *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE. 2013, pp. 48–62 (cit. on p. 13).

[Tea] Openssl Team. *TLS heartbeat read overrun (CVE-2014-0160)*. URL: `https://www.openssl.org/news/secadv_20140407.txt` (visited on 01/18/2017) (cit. on pp. 23, 46).

[Tea15] The PaX Team. *RAP: RIP ROP*. 2015. URL: `https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf` (visited on 02/02/2017) (cit. on pp. 32, 37).

[Thi16] Sam Thielman. *Yahoo hack: 1bn accounts compromised by biggest data breach in history*. 2016. URL: `https://www.theguardian.com/technology/2016/dec/14/yahoo-hack-security-of-one-billion-accounts-breached` (visited on 01/15/2017) (cit. on p. 1).

[Tic+14] Caroline Tice et al. "Enforcing forward-edge control-flow integrity in GCC & LLVM." In: *23rd USENIX Security Symposium (USENIX Security 14)*. 2014, pp. 941–955 (cit. on pp. 7, 37).

[Tra+11] Minh Tran et al. "On the expressiveness of return-into-libc attacks." In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2011, pp. 121–141 (cit. on p. 17).

[TSS15] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. "Heisenbyte: Thwarting memory disclosure attacks using destructive code reads." In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2015, pp. 256–267 (cit. on p. 30).

[V+12] Victor Van der Veen, Lorenzo Cavallaro, Herbert Bos, et al. "Memory errors: the past, the present, and the future." In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2012, pp. 86–106 (cit. on p. 2).

[Wil+16] David Williams-King et al. “Shuffler: Fast and deployable continuous code re-randomization.” In: *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*. USENIX Association. 2016, pp. 367–382 (cit. on pp. 30–32).

[XKI03] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K Iyer. “Transparent runtime randomization for security.” In: *Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on*. IEEE. 2003, pp. 260–269 (cit. on p. 4).

[Zal] Michael Zalweski. *AFL: the bug-o-rama trophy case*. URL: `http://lcamtuf.coredump.cx/afl/#bugs` (visited on 01/13/2017) (cit. on pp. 3, 4).