Memory corruption mitigation via hardening and testing

A Dissertation presented

by

László Szekeres

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

May 2017

Stony Brook University

The Graduate School

László Szekeres

We, the dissertation committe for the above candidate for the

Doctor of Philosophy degree, hereby recommend

acceptance of this dissertation

Dr. R. Sekar – Dissertation Advisor
Professor, Department of Computer Science

Dr. Scott Stoller – Chairperson of Defense
Professor, Department of Computer Science

Dr. Long Lu – Committee Member
Assistant Professor, Department of Computer Science

Dr. Franjo Ivančić – External Committee Member
Staff Software Engineer, Google
Adjunct Assistant Professor, Columbia University

This dissertation is accepted by the Graduate School

Charles Taber
Dean of the Graduate School

Abstract of the Dissertation

Memory corruption mitigation via hardening and testing

by

László Szekeres

Doctor of Philosophy

in

Computer Science

Stony Brook University

2017

Despite decades of research, memory corruption vulnerabilities continue to be at the forefront of today's security threats. There are two main avenues for automating defenses against these vulnerabilities: (a) develop automated analysis/testing techniques to find these bugs, or (b) develop software transformations that block their exploitation, or at least make them more difficult (hardening). This dissertation explores both of these avenues.

The thesis begins with a general model of memory corruption attacks and a systematic analysis of previous hardening techniques. We show that there none of the existing solutions protect against control-flow hijacks in a both effective and efficient manner. Next, we introduce a novel security policy called *code-pointer integrity* (CPI) that guarantees the protection of all code pointers in a program, thereby making control-flow hijacks impossible. We also show that it can be enforced efficiently by providing an implementation with low overhead. Some elements of this solution have already been adopted into the popular LLVM compiler.

The second part of the dissertation focuses on automated techniques for finding memory corruption bugs. Fuzzing and dynamic symbolic execution (DSE) are the two main approaches currently being used for bug finding. The key strength of fuzzing is that it is simple to use, and can generate and evaluate a very large number of test cases in a short time. Its main drawback is its lack of direction (or blindness), which means that the vast majority of test cases are not useful. It is in this regard that DSE excels: it can generate inputs that can target specific paths, providing a reliable way to increase coverage. However, targeted input generation is far slower than the test generation of fuzzers, leading to far fewer test cases being explored every second. We propose a new approach called *search-based fuzzing* (SBF) that spans the gap between the two extremes represented by fuzzing and DSE. SBF's input generation is much more targeted than today's fuzzers, yet significantly faster than symbolic execution techniques. This combination seems very promising: we show that SBF can achieve significantly more coverage than state-of-the-art tools for fuzzing and DSE.

Dedicated to

my parents, Ilona and László,
without whom I would have never started,

&

to the love of my life, María Isabel,
without whom I would have never finished.

Contents

List of Tables

List of Figures

List of Algorithms

List of Code Examples

Acknowledgments

I owe my deepest gratitude to my advisor, Prof. R. Sekar, who guided me throughout this journey. Sekar fully supported me in everything I set my mind to. He encouraged me to follow my own research ideas and kept me motivated with his invaluable insights and critical thinking. I especially thank him for being patient with me even when I was stubborn. I could not have had a better PhD advisor.

I would also like to thank my committee members, Franjo Ivančić, Long Lu, and Scott Stoller, for their time for serving on the committees of my thesis proposal and defense, and for the great feedback and insightful comments they gave.

During my PhD studies I was fortunate to spend a year as a visiting researcher at Dawn Song's group at UC Berkeley. It was an honor for me to work with her and with all other members of the group. I especially learned a lot from Lorenzo Martignoni and Stephen McCamant, and I had a great time discussing research with my many other colleagues and friends at Berkeley. I am particularly thankful for Dan Caselden for being a great friend during my stay.

I was also more than lucky to have the chance to spend a year as a research intern at Google, which also had a great influence on my research. Domagoj Babić and Franjo Ivančić were the best hosts and mentors one could wish for. I am indebted to them and to all my colleagues at Google, from whom I learned so much and who provided a fantastic environment to work in.

I share the credit of the work presented in this dissertation with Dawn Song, George Candea, Mathias Payer, R. Sekar, Tao Wei, and Volodymyr Kuznetsov. I thank them for their collaboration. I also thank all my other co-authors with whom I had the opportunity to work on other projects during my PhD.

I thank my labmates in the Secure Systems Lab: Alireza Saberi, Mingwei Zhang, Niranjan Hasabnis, Riccardo Pelizzi, Rui Qiao, Tung Tran, Wai-Kit Sze, and Yarik Markov, for their company and friendships. We had a lot of fun times together.

I would also like to thank all the wonderful friends I made during these years in Stony Brook, in California, and elsewhere, for the good times we spent together.

The most special thanks goes to my fiancée, María Isabel. Without her infinite love and support I would not have been able to finish my dissertation.

Finally, I am grateful for my parents Ilona and László, and my sister, Szilvia. They always encouraged me to pursue my dreams and supported me along the way.

This work was supported in part by grants from NSF (CNS-831298 and CNS-1319137) and ONR (N00014-15-1-2378).

Chapter 1

Introduction

Memory corruption related vulnerabilities in software written in C or C++ are one of the oldest problems in computer security. The lack of memory safety in these languages allows bugs such as *buffer overflows* and *use after frees* that attackers can exploit to maliciously alter the program's behavior, and often, take full control over the entire system [119, 119, 38, 22]. The most commonly used low-level and performance critical systems, such as operating systems, web browsers and web servers are written in C/C++ today. This means that our entire computing infrastructure for the foreseeable future is built on a codebase prone to serious vulnerabilities.

One obvious solution would be to avoid these languages in the first place and rewrite all software in type-safe alternatives. Unfortunately, often there is no such type safe language that allows the same control necessary for low-level systems programming, but even if there was, it would take decades to replace all the existing code on which our core infrastructure is built up on. This is why in this dissertation we deal with the problem of memory corruption vulnerabilities in *existing* code. In general, there are two approaches to mitigate these:

1. Bug-finding: find the bugs in the code that cause the vulnerabilities and fix them.
2. Hardening: do not find the bugs, but make them hard (if not impossible) to exploit.

To solve the problem of securing vast quantities of existing code, we need tools that test and find bugs in the code in an automated fashion and/or make it more robust against attacks without the need of manual code modifications. This dissertation makes contributions in both of these areas. The first part covers hardening techniques, and the second covers automated security testing techniques.

1.1 Software Security Hardening

Our reliance on vulnerability-prone infrastructure leads to a constant struggle between software vendors trying to defend against vulnerabilities, and the attacker community that attempts to actively find and exploit them. We call this the *eternal war in memory* between offensive research that develops new exploitation methods, and defensive research that develops new hardening techniques to make C/C++ programs safer. This arms race between offense and defense has persisted since the 80's, beginning with the appearance of the Morris worm, which was the first high-profile attack exploiting a buffer overflow vulnerability. According to the MITRE ranking [114], memory corruption bugs are considered one of the top three most dangerous software errors even today. On hacking contests like Pwn2Own or Pwnium, all major browsers (Chrome, IE, Firefox) get successfully exploited each year using memory corruption attacks.

The most common, and at the same time, most dangerous type of attack is the *control-flow hijack* attack. Attackers gain remote control of victim systems using this type of attack. Over the last 30 years, many defenses have been developed against this and related exploits. Some of them are deployed in commodity systems and compilers, including stack cookies [60], exception handler validation (SafeSEH), Data Execution Prevention (DEP) [166] and Address Space Layout Randomization (ASLR) [130]. These techniques make it harder to exploit memory corruption bugs. Nevertheless several attack vectors continue to be effective despite these protections. In particular, return-oriented programming (ROP) [119, 149, 38, 22, 164, 136], information leaks [156, 148], heap spraying, and the prevalent use of user scripting and just-in-time compilation [21] allow attackers to take control over systems despite all these protections.

A multitude of defense mechanisms have been proposed to overcome one or more of the possible attack vectors. Yet most of them are not used in practice, due to one or more of the following factors. The *performance* overhead of protections is often deemed high, or the approach is not *compatible* with all currently used features (e.g., in legacy programs). Another barrier is when manual source code modification is necessary to apply the protection, or to make it compatible with existing code. A final reason can be that the approach is not *robust* enough to offer complete protection against an attack.

With all the diverse attacks and proposed defenses it is hard to see how effective and how efficient different solutions are and how they compare to each other and what the primary challenges are. Therefore, Chapter 2 of the dissertation analyses and evaluates previously proposed approaches in a systematic fashion. We set up a general model for memory corruption vulnerabilities and exploitation techniques. Defense techniques are then classified by the exploits they mitigate and by the particular phase of exploit they try to inhibit. We evaluate these techniques based on the protection's strength, performance and compatibility.

Using our model and evaluation criteria, we identified two subcategories of control-flow hijacking defense where significant advances were possible, and developed new and practical solutions in each category. The first among these was CCFIR [185] a hardening technique for Windows binaries to enforce *control-flow integrity* (CFI). CFI, introduced by Abadi et al. [1], is a policy that restricts control-flow transfers susceptible for hijacks to a limited set of allowed destinations, in order to detect attacks. Our solution was the first that demonstrated the feasibility of transforming large COTS binaries to protect them with CFI, while providing excellent performance. In this dissertation we do not cover CCFIR in detail. Instead, we focus on describing our second technique, which *prevents* all control-flow attacks.

For this solution, we observe that all control-flow hijack attacks need to corrupt a code pointer, namely, the pointer used to determine the target of the hijack. Therefore, we introduce a new, low-level security policy called *code-pointer integrity* (CPI), which ensures that no code pointer can be corrupted, thereby rendering control-flow hijacks impossible. In Chapter 3, we show how to enforce this policy efficiently and effectively, so that the integrity of all code pointers in a program is guaranteed. The only existing approach that gives the same guarantee is enforcing *memory safety* for existing C/C++ programs, but such state-of-the-art techniques incur more than 2× overhead [116, 117]. The high overhead is because we need to keep track of extra metadata for each pointer, which is used to check whether the pointers are used in a safe way.

The key idea of our approach is to enforce *memory safety* selectively, so that we only protect the minimum set of pointers necessary to enforce *code-pointer integrity*. We use static analysis to identify the set of pointers that must be protected in order to guarantee memory safety for code pointers. This set includes pointers that are code pointers themselves, and other data pointers that can be used to directly or indirectly access code pointers. We store all protected pointers together with their metadata in a safe region. This region is isolated from the rest of the address space (e.g., via hardware protection) and can only be accessed by our instrumentation.

With our technique, in typical programs, only a small fraction of the pointer operations need to be instrumented and checked (6.5% for SPEC CPU2006 programs). CPI fully protects the program against all control-flow hijack attacks that exploit program memory bugs. It requires no changes to how programmers write code, since it automatically instruments pointer accesses at compile time. Further, it achieves low overhead by selectively instrumenting only those pointer accesses that are necessary and sufficient to guarantee the integrity of all code pointers. Finally, our approach can also be used to protect data variables as well, e.g., to selectively protect sensitive information like the process UIDs in a kernel.

We also introduce *code-pointer separation* (CPS), a relaxed variant of CPI that is better suited for code with abundant virtual function pointers. In CPS, all code pointers are securely stored in a safe region, but pointers used to access code pointers indirectly are left unprotected (such as pointers to C++ objects that contain virtual functions). Unlike CPI, CPS may allow certain control-flow hijack attacks, but it still offers stronger guarantees than CFI and incurs negligible overhead.

Our experimental evaluation shows that our proposed approach imposes sufficiently low overheads. For example, measured on SPEC CPU2006 benchmark, CPS incurs an average overhead of 1.2% on the C programs and 1.9% for all C/C++ programs. CPI incurs on average 2.9% overhead for the C programs and 8.4% across all C/C++ programs. CPI and CPS are effective: they prevent 100% of the attacks in the RIPE benchmark and attacks [69, 46, 33] that bypass CFI approaches, and other hijack defenses. We compile and run with CPI/CPS a complete FreeBSD distribution along with over a 100 widely used packages, demonstrating that the approach is practical.

In Part I of this dissertation we make the following contributions:

1. We develop a general model of memory corruption attacks and match low-level security policies with the exploit steps of this model, showing which policy mitigates which steps. The model is not only used to clarify what attack vectors are left unprotected by currently used and previously proposed protections, but it also helped us identify a new policy, which is the basis of our next contribution.

2. We introduce a new low-level security policy, called *code-pointer integrity* (CPI), that prevents all control-flow hijack attacks. We also introduce a relaxed version of this policy, called *code-pointer separation* (CPS), which provides stronger security guarantees than *control-flow integrity* but at negligible cost.

3. We provide an efficient compiler-based implementation of CPI and CPS for unmodified C/C++ code. Our implementation contains a return address protection called SafeStack, which can also be used separately. SafeStack ensures the integrity of return addresses with negligible or zero overhead and it is already part of the widely used LLVM/Clang compiler.

1.2 Software Security Testing

Hardening techniques alone do not fully solve the general problem of memory corruption bugs in existing software. None of the existing approaches provide guaranteed protection against all types of attacks in a practical and compatible way, with low performance overhead. CPI can prevent control-flow hijacks with a low cost, but data-only attacks [79] that corrupt only data variables remain possible. Therefore, finding and fixing memory corruption bugs is equally important today as making them hard to exploit.

There are two main automated testing approaches for finding memory corruption bugs: fuzzing and dynamic symbolic execution (DSE). Fuzzing [125] involves running the program under test with randomly mutated inputs and detecting crashes. It is simple to use, requiring little user effort. Fuzzing introduces very little (runtime or memory) overhead to the test runs, so it scales well. These factors make fuzzing the popular choice for most security testing practitioners. Many of the infamous vulnerabilities of the past few years, such as Heartbleed [41], Shellshock [181], and Stagefright [55], were all found with fuzzing. Despite these successes, fuzzing tools are rather limited in their ability to cover large parts of the code. Simple conditions, such as the one shown in the code snippet below, can take billions of attempts to get through:

```c
void fuzzme(int input) {
  if (input == 0xbadc0de) { /* some vulnerable code */ }
}
```

This is because fuzzing techniques mutate their input *blindly*, not considering program logic or structure. In contrast, DSE [32] generates inputs by directly targeting specific conditions to "punch" through. The program is run with a symbolic input in order to generate a formula that captures the constraints necessary to take a particular path in the program. These constraints are then solved by a constraint solver to get a concrete input that traverses the targeted path. By capturing these detailed path conditions, and solving them using powerful solvers, DSE systems are able to generate inputs that can easily get through relatively complex conditions.

In theory, the systematic approach embedded in DSE systems should be able to exercise most program paths, and thereby achieve high code coverage. In practice, however, the costs and limitations of DSE can prevent it from achieving this potential. In their recent SoK paper, Shoshitaishvili et al. [151] report that fuzzing tools identify almost three times as many vulnerabilities as DSE techniques. They go on to say "In a sense, this mirrors the recent trends in the security industry: symbolic analysis engines are criticized as impractical while fuzzers receive an increasing amount of attention. However, this situation seems at odds with the research directions of recent years, which seem to favor symbolic execution."

Chapter 4 of the dissertation provides a survey of automated testing research, where we discuss in more detail the above mentioned costs and trade-offs, and offer more insight how these techniques relate to each other. We provide an in-depth analysis of fuzzing, DSE and related techniques, such as constraint solving and search-based software testing [109]. We develop a uniform framework to relate the different techniques and analyze their strengths and weaknesses and also identify the main unsolved challenges.

Trying to harness the best of fuzzing and DSE, researchers have developed hybrid systems that employ fuzzing to generate most test inputs, and using DSE selectively when the fuzzer gets "stuck." During the recent DARPA Cyber Grand Challenge competition most of the teams, including the top three used this approach [121, 11, 70, 155]. Although such a hybrid approach can help avoid the worst outcomes, it does not eliminate or solve their underlying problems. In particular, these approaches do not fundamentally address the lack of direction in fuzzing tools, nor do they overcome all the challenges of applying DSE on large code bases. As a result, the problem of developing an automated testing approach that is directed yet scalable has remained open. In Chapter 5 we present *search-based fuzzing* (SBF), a new approach that fills this gap.

Automated test generation tools make a trade-off between (a) the computational resources needed to generate a new input, and (b) the likelihood of increased coverage due to this input. Fuzzing, at one end of the spectrum, spends very little time on input generation, but these randomly generated inputs have a very low probability of increasing coverage. At the opposite end of the spectrum, DSE generates inputs with a high probability of increasing coverage, but to do so, it expends a lot of resources in generating each input. In this thesis we present a novel approach called *search-based fuzzing* (SBF) that falls in between these two approaches, thus representing a more favorable trade-off between the two extremes.

Like DSE, SBF is *directed:* it targets specific conditional branches to take in order to increase coverage. Like fuzzers, it needs only a light-weight source instrumentation, thereby avoiding the complexity of DSE, and the performance costs of constraint solvers.

Unlike fuzzers that tend to operate blindly, SBF targets its input mutations in order to reach a specific branch condition. It uses *data-flow tracking* (DFT) to determine ***what*** parts of the input affect the targeted branch condition, and targets its mutations to those bytes. It then uses a *stochastic local search* (SLS) [78] to determine ***how*** to mutate these bytes in order to take the targeted branch. Both DFT and SLS are achieved using a light-weight instrumentation, thereby enabling SBF to retain the performance benefits of fuzzing.

By combining the key strengths of DSE and fuzzing, namely, directionality and performance, SBF can be more effective than either class of approaches. For example, in the "magic value" example mentioned above, a blind fuzzer needs more than two billion executions on average to generate an input that satisfies the condition on a 32-bit value. In contrast, SBF reduces this to about 32 attempts using SLS. As a more complex example, consider the Adler checksum algorithm that is used in `zlib`, `rsync`, and many other systems. Our measurements show that blind fuzzers like AFL need several hours, if not days, to generate an input with a specific Adler checksum. In contrast, SBF completes this task in seconds using its combination of SLS and DFT. As a result, SBF can achieve higher coverage in shorter time than state-of-art fuzzing (or DSE) tools, and consequently, find more bugs in a much shorter time.

Our evaluation shows that SBF is more efficient and effective in increasing coverage than AFL, LibFuzzer, or KLEE. Testing `libxml`, it reaches 2× higher coverage in a few minutes than the other tools after hours, while with `libpng` it saturates 4-6× higher than KLEE and LibFuzzer. Testing the `base64` utility with a large number of artificially injected vulnerabilities in it, SBF finds all bugs under a minute, while other tools find only a subset of bugs in 5 hours.

In Part II of this dissertation we make the following contributions:

1. We provide a survey of existing automated software security testing research and show how fuzzing, dynamic symbolic execution, constraint solving, and search-based software testing relate to each other through a uniform framework. We compare different test generation techniques that can be used to find low-level bugs by analyzing their strength and weaknesses.

2. We introduce *search-based fuzzing*, a new automated test generation technique, based on stochastic local search. The approach builds on instrumenting compare instructions in the target program and has the following three main components:

    - *search target identification*, used to select which parts of the program to target, and which tells what parts of the input to mutate to reach those targets.

- *local search based mutation strategy,* which indicates how to mutate the relevant input bytes to reach the selected target program point. We propose a special *stochastic local search* based on the Markov Chain Monte Carlo (MCMC) algorithm.
- *test suite inflation and deflation* technique, which overcomes the limitations of edge coverage metric while avoiding path explosion.

3. Finally, we also release the implementation of SBF as an open-source tool.

Part I

Program Hardening against Memory Corruption Attacks

Chapter 2

The Landscape of Memory Corruption

This chapter systematizes the state-of-the-art knowledge about memory corruption related bugs, attacks, and protection mechanisms. First we set up a general model of attacks exploiting memory corruption bugs. Next, we define low-level security policies in the context this attack model, showing which policy can stop which attack. Finally, we analyze protection mechanisms, implementing these policies.

We describe the primary reasons why attacks still succeed on today's systems. Our analysis identifies weaknesses not only in currently deployed techniques, but also in previously proposed protections, that enforce stricter policies. We also discuss the reasons why these stricter protection mechanisms are not deployed. We find that in order to achieve wide adoption, protection mechanisms must support a multitude of features and satisfy a host of requirements. Especially important is performance, as experience shows that only low-overhead solutions get deployed.

2.1 Anatomy of Memory Corruption Exploits

To solve the problem of memory error based attacks, we first need to understand the process of carrying out such an exploit. In this section we set up a step-by-step memory exploitation model. We will base our discussion of protection techniques and the policies they enforce on this model. Figure 2.1 shows the different steps of exploiting a memory error. Each white rectangular node represents a building block towards successful exploitation and the oval nodes on the bottom represent a successful attack. Each rhombus represents a decision between alternative paths towards the goal. While control-flow hijacking is usually the primary goal of attacks, memory corruption can be exploited to carry out other types of attacks as well.

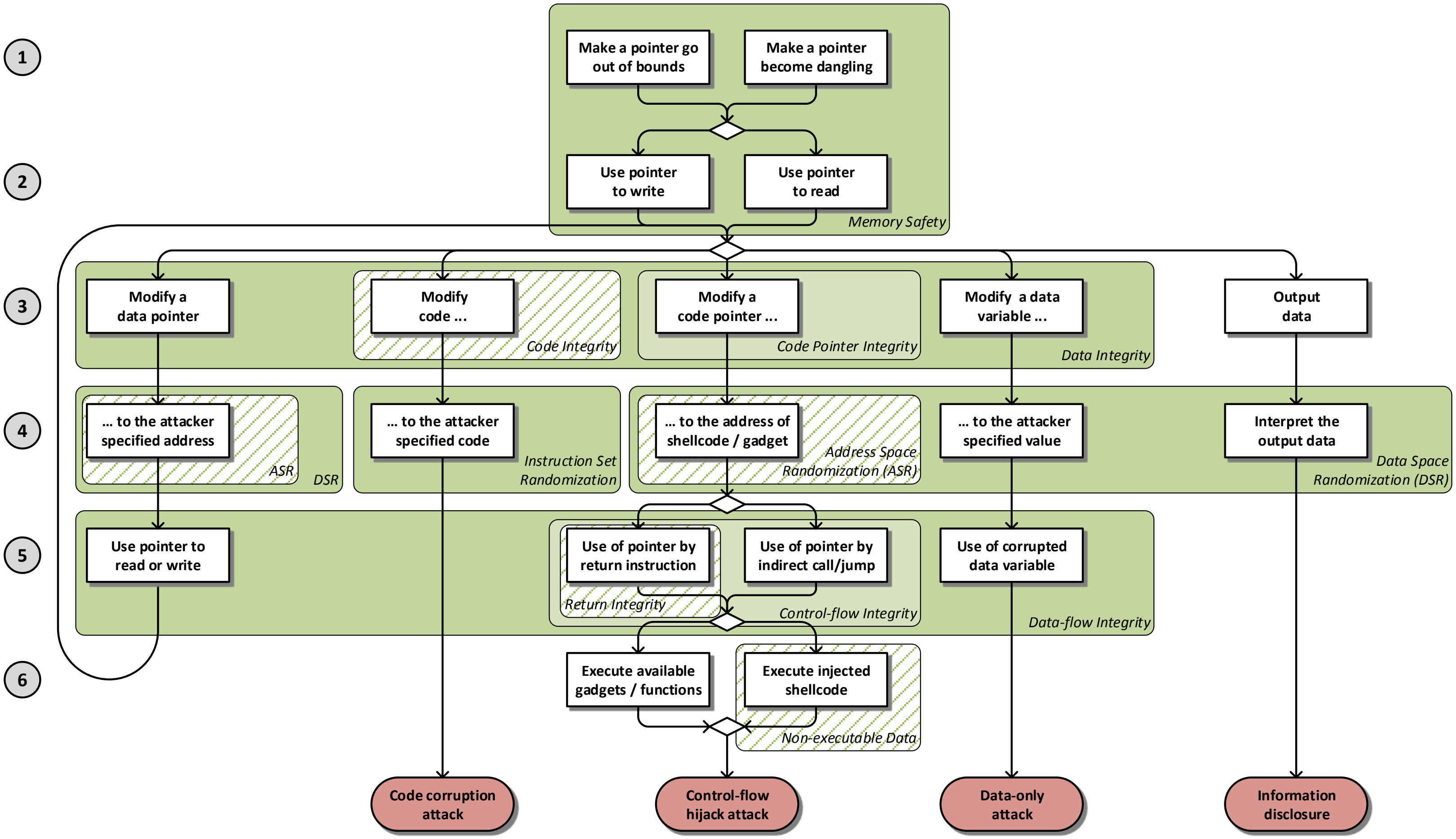

1
2
3
4
5
6
Make a pointer go out of bounds
Make a pointer become dangling
Use pointer to write
Use pointer to read
Memory Safety
Modify a data pointer
Modify code ...
Code Integrity
Modify a code pointer ...
Code Pointer Integrity
Modify a data variable ...
Data Integrity
Output data
... to the attacker specified address
ASR
DSR
... to the attacker specified code
Instruction Set Randomization
... to the address of shellcode / gadget
Address Space Randomization (ASR)
... to the attacker specified value
Interpret the output data
Data Space Randomization (DSR)
Use pointer to read or write
Use of pointer by return instruction
Return Integrity
Use of pointer by indirect call/jump
Control-flow Integrity
Use of corrupted data variable
Data-flow Integrity
Execute available gadgets / functions
Execute injected shellcode
Non-executable Data
Code corruption attack
Control-flow hijack attack
Data-only attack
Information disclosure

Figure 2.1: Attack model demonstrating four exploit types and policies mitigating the attacks in different stages

2.1.1 Memory Corruption

Every exploit starts by triggering a memory error. The first two steps of an exploit in Figure 2.1 cover the initial memory error. The first step makes a pointer *invalid*, while the second step dereferences the pointer, thereby triggering the error. A pointer can become invalid by going out of the bounds of its pointed object or when the object gets deallocated. A pointer pointing to a deleted object is called a *dangling* pointer. Dereferencing an out-of-bounds pointer causes a so called *spatial error*, while dereferencing a dangling pointer causes a *temporal error*.

In Step 1, an attacker may force a pointer out of bounds by exploiting programming bugs. For instance, by triggering an *allocation failure* which is unchecked, the pointer can become a *null pointer* (null-pointer dereferences are especially dangerous in kernel-space [88]). By excessively incrementing or decrementing an array pointer in a loop without proper bound checking, a *buffer overflow/underflow* will happen. By causing *indexing bugs* where an attacker has control over the index into an array, but the bounds check is missing or incomplete, the pointer might be modified to any address. Indexing bugs are often caused by integer related errors like an *integer overflow*, truncation or signedness bug, or incorrect pointer casting.

As an alternative, the attacker may cause a pointer to *dangle*, e.g., by exploiting an incorrect exception handler that deallocates an object without reinitializing the pointers to it. Temporal memory errors are called *use-after-free* vulnerabilities because the dangling pointer is dereferenced (used) after the memory area it points to has been returned to the memory management system (freed). Most of the attacks target heap allocated objects, but pointers to a local variable can also "escape" from local scope when assigned to a global pointer. Such escaped pointers become dangling when the function returns.

Next, we show how either an out-of-bounds or a dangling pointer can be exploited to execute any third step in our exploitation model when the invalid pointer is *read* or *written* in Step 2. The third step is either the corruption or the disclosure of some internal data.

When a value is *read* from memory into a register by dereferencing a pointer controlled by the attacker, the value can be corrupted. Consider the following jump table where the function pointer defining the next function call is read from an array without bounds checking.

```
func_ptr jump_table[3] = {fn_0, fn_1, fn_2};
jump_table[user_input]();
```

The attacker can make the pointer point to a location under his or her control and divert control flow. Any other variable read indirectly can be vulnerable.

Besides data corruption, reading memory through an attacker specified pointer leaks information if that data is included in the output. The classic example of this attack is the `printf` *format string bug*, where the format string is controlled by the attacker.

By specifying the format string the attacker creates invalid pointers and reads (and writes) arbitrary memory locations.

```
printf(user_input); // Input "%3$x" prints the 3rd integer on the stack.
```

If an attacker controlled pointer is used to *write* the memory, then any variable, including other pointers or even code, can be overwritten. Buffer overflows and indexing bugs can be exploited to overwrite sensitive data such as a return address or virtual table (vtable) pointer. Corrupting the vtable pointer is an example of the backward loop in Figure 2.1. Suppose a buffer overflow makes an array pointer out of bounds in the first round that is exploited (in Step 3) to corrupt a nearby vtable pointer in memory in the second round. When the corrupted vtable pointer is dereferenced (in Step 2), a bogus virtual function pointer will be used. It is important to see that with one memory error, more and more memory errors can be raised by corrupting other pointers. Calling `free()` with an attacker controlled pointer can also be exploited to carry out arbitrary memory writes [85].

Write dereferences can be exploited to leak information as well. For instance, the attacker is able to leak arbitrary memory contents in the following code by corrupting the `err_msg` pointer:

```
printf("%s\n", err_msg);
```

Temporal errors, when a dangling pointer is dereferenced in Step 2, can also be exploited. A constraint for exploitable temporal errors is that the memory area of the deallocated object (the old object) is reused by another object (new object). The type mismatch between the old and new object can allow the attacker to access unintended memory. Attacks can be carried out by both reading and writing through a dangling pointer. We discuss both cases.

Let us consider first reading through a dangling pointer pointing to a new object whose contents are controlled by the attacker. Consider for instance when a virtual function of the old object is called and the virtual function pointer is looked up. In this case the contents of the new object will be interpreted as the vtable pointer of the old object. This allows the corruption of the vtable pointer, comparable to exploiting a spatial write error, but in this case the dangling pointer is only dereferenced for a read. This is a common pattern in real world exploits [71]. An additional aspect of this attack is that the new object may contain sensitive information that can be disclosed when read through the dangling pointer of the old object's type.

Now let us consider cases when the attack can write through dangling pointers. This can be exploited similarly to an out of bounds pointer, by corrupting other pointers or data inside the new object. When the dangling pointer is an escaped pointer to a local variable and points to the stack, it may be exploited to overwrite sensitive data, such as a return address. A special case of using dangling pointers is a *double-free*, where the dangling pointer is used to call `free()` again. In this case, the attacker controlled contents of the new object will be interpreted wrongly as heap metadata, which is also exploitable for arbitrary memory writes [85].

In summary, memory errors allow an attacker to read and modify the program's internal state in unintended ways. We showed that *any* combination of the first two steps in our memory exploitation model can be used both to corrupt internal data and to gain access to sensitive information. Furthermore, more memory errors can be triggered by corrupting other pointers. Programming bugs underpinning these errors, such as buffer overflows and double-frees, are common in C/C++. When developing in such low-level languages, both bounds checking and memory management are fully the programmers responsibility. Unfortunately, both tasks are highly error prone.

The above described errors are a violation of the *memory safety* policy. C and C++ are inherently memory unsafe. According to the C/C++ standards, writing an array beyond its bounds, dereferencing a null-pointer, or reading an uninitialized variable result in *undefined* behavior. Since finding and fixing all the programming bugs is infeasible, we need automatic solutions to enforce memory safety in existing programs or stop attacks in their later phases. The policy mitigating a given set of attack steps is represented in Figure 2.1 by a colored area surrounding the white boxes. We discuss approaches that try to stop any exploit in the first (two) steps by enforcing *memory safety* in Section 2.5.1. In the following subsections we discuss the steps of different exploit paths and identify the policies mitigating the given step.

2.1.2 Exploit Avenues

Code Corruption

The most obvious way to modify the execution of a program is to use one of the above mentioned bugs to overwrite the program code in memory. The code integrity policy prevents program code from being written. *Code integrity* can be achieved if all memory pages containing code are set read-only, which is supported by all modern processors. Unfortunately, *code integrity* does not support self-modifying code or Just-In-Time (JIT) compilation. Today, every major browser includes a JIT compiler for JavaScript or Flash. For these use-cases, *code integrity* cannot be fully enforced because there is a time window during which the generated code is on a writable page.

Control-flow Hijack

When *code integrity* is enforced [188], hijacking the control flow becomes the typical goal, making this attack by far the most common. This attack avenue consists of several steps, specifically steps 3 to 6 in the middle of Figure 2.1. We describe each of these steps and the policies that can block or mitigate a given step.

Having an invalid pointer after Step 2, the next move towards hijacking is to corrupt a code pointer in Step 3. Here we introduce a new policy, called code-pointer integrity that *prevents* this step. No previous work has tried to stop a control-flow hijack in this step. This policy prohibits the corruption of code pointers through either a spatial or a temporal error. We will discuss this policy in detail in Chapter 3 where we also show how it can be enforced efficiently.

Moving to the next attack step, suppose the attacker can access and modify a return address due to a buffer overflow at this point. For a successful exploit, the attacker needs to know the correct target value (i.e., the address of the payload) as well. We represent this as a separate fourth step in Figure 2.1. If the target of the control-flow hijack (the code address to jump to) is not fixed, the attacker cannot specify the target and the attack fails at this step. This property can be achieved by introducing entropy to memory addresses using *address space randomization* (ASR). We discuss techniques randomizing the address space in Section 2.4.1.

Suppose a code pointer (e.g., a function pointer) has been successfully corrupted in the first four steps. The fifth step is that the CPU needs to load the corrupted pointer into its instruction pointer register. The instruction pointer can only be updated indirectly by executing an indirect control-flow transfer instruction, e.g., an indirect function call, indirect jump or function return instruction. Diverting the execution from the control flow defined by the source code is a violation of the *control-flow integrity* (CFI) policy. In Section 2.6.2, we cover protections that enforce different CFI policies by *detecting* corruption at indirect control transfers.

The final step of a control-flow hijack exploit is the execution of attacker specified malicious code. Classic attacks injected so-called shellcode into memory, and diverted execution to this piece of code. This kind of exploitation is prevented by the *non-executable data* policy which can be enforced using the executable bit for memory pages to make data memory pages, like the stack or the heap, non-executable. A combination of *non-executable data* and *code integrity* results in the W⊕X (Write XOR Execute) [166] policy, stating that a page can either be writable or be executable, but not both.

Practically all modern CPU support setting non-executable page permissions, so combined with non-writable code, enforcing W⊕X is cheap and practical. However, in the case of JIT compilation or self-modifying code, W⊕X cannot be fully enforced. For the sake of completeness, we note that another randomization approach, *instruction set randomization* (ISR) can also mitigate the execution of injected code or the corruption of existing code by encrypting it. Since hardware support for page permissions became available however, the high performance overhead of ISR makes it a less appealing choice, as compared to the cheap and ubiquitous W⊕X.

To bypass the *non-executable data* policy, attackers can reuse existing code in memory. The reused code can be an existing function ("return-to-libc" attack) or small instruction sequences (gadgets) found anywhere in the code that can be chained together to carry out useful (malicious) operations. This approach is called Return Oriented Programming (ROP), because the attack chains the execution of functions or gadgets using the ending return instructions. Jump Oriented Programming (JOP) is the generalization of this attack which leverages indirect jumps as well for chaining. There is no policy which can stop the attack at this point, since the execution of valid and already existing code cannot be prevented. Recent research focuses on mitigating techniques against code reuse only. Researchers propose techniques to eliminate useful code chunks (for ROP) from the code by the compiler [126], or by binary rewriting [129]. While these solutions make ROP harder, they do not eliminate all useful gadgets, and they do not prevent re-using complete functions. For these reasons we will not cover these techniques in more detail.

We classify a control-flow hijacking attack as successful when the attacker-specified code starts to execute. To carry out a meaningful attack, the attacker usually needs to make system calls, and may need high level permissions (e.g., file access) as well. We will not cover higher-level policies which only confine the attacker's access, including permissions, mandatory access control or sandboxing policies enforced by SFI [169], XFI [59] or Native Client [177]. These policies can limit the damage that an untrusted program (or plugin) or an attacker can cause *after* compromising a trusted program. Our focus is preventing the compromise, we do not discuss post-exploit damage mitigation steps.

Data Corruption

Hijacking control flow is not the only possibility for a successful attack. In general, the attacker's goal is to maliciously modify the program logic to gain more control, to gain privileges, or to steal information. This goal can be achieved without modifying data that is expliclity related to control flow. Consider, for instance, the modification of the isAdmin variable via a buffer overflow after logging into the system with no administrator privileges.

```
bool isAdmin = false;
...
if (isAdmin) {
  // Do privileged operations.
}
```

These program specific attacks are also called *non-control-data attacks* [39] since neither code nor code pointers (control data) are corrupted. The target of the corruption can be any security critical data in memory, e.g., configuration data, the representation of the user identity, keys, or anything else. It has also been shown that through the corruption of only data and data pointers, Turing-complete computation can be carried out [79]. Note that such attack typically requires a long series of data pointer corruption through the back loop show in Figure 2.1.

The steps of this attack are similar to the previous one, except for the target of corruption. Here, the goal is to modify some variable through the corrupted point in Step 3. We call the policy that prevents this step *data integrity*. This policy naturally includes *code integrity* and *code-pointer integrity*. *Data integrity* approaches try to prevent the corruption of data by enforcing only some approximation of *memory safety*. We cover techniques enforcing such policies under 2.5.2.

As in the case of code pointers, the attacker needs to know what should replace the corrupted data. Acquisition of this knowledge can be prevented by introducing entropy into the representation of *all* data using *data space randomization*. *Data space randomization* techniques use encryption or obfuscation of data to mitigate this step, and we cover them in Section 2.4.2.

Similar to code pointer corruption, data-only attacks will succeed as soon as the corrupted variable is used. Using the running example, the `if (isAdmin)` statement has to be successfully executed without *detecting* the corruption. As the generalization of *control-flow integrity*, the use of *any* corrupted data is a violation of *data-flow integrity*. We cover enforcing this policy under Section 2.5.3.

Information Disclosure

We showed that any type of memory error might be exploited to disclose memory contents that would otherwise be excluded from the output. This is typically used to circumvent probabilistic defenses based on randomization and secrets. Real-world exploits bypass ASLR using information leaks [148, 80]. Among the covered policies, *memory safety* and full *data space randomization* prohibit leaking information through corrupted pointers. We will discuss in Section 2.4 how effective *data space randomization* is and how information leakage can be used to bypass other probabilistic techniques which build on secrets.

2.2 Currently Deployed Protections and Real World Exploits

The most widely deployed protection mechanisms are stack smashing protection, DEP, which is also called W⊕X, and ASLR (the most common form of ASR). The Windows platform for instance, also offers some special mechanisms, e.g., for protecting heap metadata and exception handlers (SafeSEH and SEHOP).

Stack smashing protection [60] detects buffer overflows of local stack-based buffers, which overwrite the saved return address. By placing a random value (called cookie or canary) between the return address and the local buffers at function entries, the integrity of the cookie can be checked before the return of the function and thus the overflow can be detected. SafeSEH and SEHOP also validate exception handler pointers on the stack before they are used, which makes them, together with stack cookies, a type of *control-flow integrity* solution. These techniques provide the weakest protection: they place checks only before a small subset of indirect jumps, checking the integrity of only some specific code pointers, namely saved return addresses and exception handler pointers on the stack. Furthermore, the checks can be bypassed. Cookies, for instance, can detect a buffer overflow attack but not a direct overwrite that exploits an indexing error.

DEP/W⊕X can protect against code injection attacks, but does not protect against code reuse attacks like ROP. ROP exploits can be generated automatically [142], and large code bases like the standard C library usually provide enough gadgets for Turing-completeness [164, 136]. ASLR provides the most comprehensive protection as the most widely deployed *address space randomization* technique. It can randomize the locations of various memory segments, including data and code, so even if the attacker wants to reuse a gadget, its location will be random. While some ASLR implementations have specific weaknesses (e.g., code regions left in predictable locations and brute-force attacks due to low entropy), a more fundamental attack against it is based on information leakage [156].

As described in the attack model of the previous section, any memory corruption error can be converted into an information leak vulnerability, which can be used to obtain current code addresses. The leaked addresses are needed to construct the final exploit payload. When attacking remote targets (i.e., servers), getting back this information used to be very challenging. Today, however, it is not a problem for a number of client targets. Web browsers, PDF viewers and office applications run user controlled scripts (JavaScript, ActionScript, VBScript), which can be used to dynamically construct exploit payloads at run-time on the target machine.

Table 2.1 lists some exploits published by VUPEN [71], which use information leaks and ROP to bypass ASLR and W⊕X. In all examples, the control flow is hijacked at an indirect call instruction (after corrupting a function pointer or the vtable), so stack cookies are not an issue. In all cases, address leakage is achieved by exploiting an arbitrary memory write in order to corrupt another pointer, which is read later. One common way of leaking out memory contents is by overwriting the length field of a JavaScript string object before reading it out in the user script. As shown in the last column, in case of browser targets, user scripting is used to leak current addresses and to construct the exploit, while in case of ProFTPD, the leaked information is sent back on the network by corrupting a pointer to a status message.

2.3 Evaluation Criteria of Protection Mechanisms

Protection mechanisms implement security policies. The previously identified policies can be divided into two main categories: probabilistic and deterministic protection. Probabilistic solutions, e.g., *instruction set randomization*, *address space randomization*, or *data space randomization*, build on randomization or encryption. All other approaches enforce a deterministic *safety policy* by implementing a low-level reference monitor [140]. A reference monitor observes the program execution and halts it whenever it is about to violate the given security policy.

Reference monitors can be coarse-grained or fine-grained. Coarse-grained reference monitors enforce higher level policies, such as file system permissions, and are imple-

CVE ID	Software	Vulnerability	Address leakage	User scripting
CVE-2011-0609	Adobe Flash	JIT type confusion	Read an IEEE-754 number	ActionScript
CVE-2012-0003	Windows Multimedia Library	Heap buffer overflow	Read a string after overwriting its length	JavaScript
CVE-2011-4130	ProFTPD	Use-after-free	Overwrite "Transfer Complete" message	none
CVE-2012-0469	Mozilla Firefox	Use-after-free	Read a string after overwriting its length	JavaScript
CVE-2012-1889	Windows XML Core Services	Uninitialized pointer	Read as a RGB color	JavaScript
CVE-2012-1876	Microsoft IE 10	Heap buffer overflow	Read a string after overwriting its length	JavaScript

Table 2.1: Exploits defeating both DEP and ASLR using ROP and information leaks

mented in the kernel (e.g., system calls). The low-level security policies that we focus on (e.g., *memory safety* or *control-flow integrity*) are enforced by fine-grained reference monitors. These can be implemented either in hardware or as an *inlined reference monitor* (IRM). For instance, the W⊕X policy (*code integrity* and *non-executable data*) is supported by most modern processors. Hardware support for a security policy results in negligible overhead. Other policies typically lack hardware support; for those IRMs can be used, where the reference monitor is embedded into the code through instrumentation. Note that hardware manufacturers usually add new security features only after their software implementation have already been proven useful. This is why our focus is on software based solutions (IRMs), which transform existing programs to enforce various policies.

The instrumentation can be done *dynamically* or *statically*. *Dynamic (binary) instrumentation* (DBI) tools, such as Valgrind [120], PIN [103], DynamoRIO [29], insert safety checks into unsafe code at run-time. While arbitrary transformations can be done via DBI, for most applications its overhead can be very high [186]. For this reason DBI based tools are mainly used for debugging purposes and not for production code. On the other hand, *static instrumentation* inlines reference monitors statically. This can be done at compile time on the source code or internal representation level, or on the binary using *static binary rewriting*. Static instrumentation is usually more efficient than dynamic, as the code transformation is not carried out at run-time, but before. Thus, the primary focus of this chapter is on static approaches.

Next, we discuss the main properties and requirements for solutions enforcing low-level policies.

2.3.1 Protection Strength

Enforced policy

The *strength* of a protection mechanism (or implementation) is determined by the policy it enforces. The exact policy that a solution enforces determines its *effectiveness*. Different techniques enforce different types of policies (e.g., *memory safety* or *data-flow integrity*) and at different levels. The practical utility of a given policy can be described by the attacks it can protect against, i.e., out of the four attack types that we identified. The *accuracy* of a mechanisms is determined by the relation between false negatives and false positives.

False negatives

The possibility of false negatives (protection failures) depends on the definition of the policy. For probabilistic approaches, the probability of a successful attack is always non-zero, while it can be zero for deterministic solutions. Recall from Section 2.2 that the secrets underlying probabilistic protections can not only be guessed, but also leaked.

False positives

False positives, e.g., unnecessary crashes or alarms, can be an even bigger issue than false negatives in practice. Causing faults during normal operation is unacceptable in production environments. Besides inaccuracies in the implementation of a policy, compatibility issues can also cause false alarms. Therefore, avoiding false positives is an important requirement for any practical solution.

2.3.2 Performance

Performance overhead

The cost of a solution is primarily determined by the performance overhead it introduces. Beside security, the most important requirement is speed.

To measure performance, both CPU-bound and I/O-bound benchmarks can and should be used. Mechanisms enforcing low-level policies usually instrument instructions such as pointer accesses. For these tools CPU-bound benchmarks, such as SPEC [74], are typically more challenging, because I/O-bound benchmarks spend more time in the kernel, reducing the impact of the relative user-space CPU overhead. For other types of instrumentation the opposite might be true, but some proposals might report good scores with selected benchmark programs or with I/O-bound server applications, while their overheads are much higher if measured using CPU-bound benchmarks. We recommend that protection approaches considered for wide adoption target CPU-bound client-side programs as well, as these are the primary targets of today's attacks.

Our comparison analysis in Section 2.7 shows that techniques introducing an overhead larger than roughly 10% do not tend to gain wide adoption in production environments. Some believe the average overhead should be less than 5% in order to get adopted by industry. For example, the Microsoft BlueHat Prize [111], a contest of memory corruption vulnerability mitigation techniques, only accepted prototypes with less then 5% overhead.

Memory overhead

Inline monitors often introduce and propagate some kind of metadata, which can introduce significant memory overhead as well. Some protection mechanisms (especially the ones using shadow memory) can even double the space requirement of a program. In case of most applications, however, this is much less of an issue than runtime performance.

2.3.3 Compatibility

Unmodified source code support

We call an approach *source compatible* (or source agnostic) if the application's source code does not need manual modifications in order to profit from the protection. The necessity of even minimal human intervention or effort makes a solution not only unscalable, but too costly as well. Most experts from the industry consider solutions which require porting or annotating the source code to be impractical.

Unmodified library support

Dynamically-linked libraries are used in most operating systems, so it is important to support them. Supporting *unmodified* libraries as well is another important practical requirement. Transformed programs should be able to link and use legacy libraries. We call a protection mechanism that ensures interoperability with unmodified binary modules *binary compatible.* Using unprotected libraries may leave parts of the program exploitable, but allows incremental deployment. Also, for instance on the Windows platform, system libraries are integrity protected and thus cannot be easily changed.

Modularity support

Often times all the code that an application will use is not available at program transformation time. Modularity support means that the protection tool should be able to transform one part of the code at a time (e.g., one compilation unit or one library) without having access to the whole program. For a compiler based solution, supporting separate compilation is important, as most build systems handle one compilation unit at a time. Further, the tool should be able to harden a main executable or a dynamic library separately, without knowing ahead of time which module will use which other modules.

2.4 Probabilistic Methods

Probabilistic methods rely on randomization and secrets. There are three main approaches: *instruction set randomization*, *address space randomization*, and *data space randomization*. Figure 2.1 shows that *instruction set randomization* (ISR) [87] mitigates attacks based on code corruption and injection of shellcode. Most modern processors support W⊕X that block injection attacks, thus obviating the need for ISR. *Address space randomization* (ASR) mitigates attacks by randomizing the location of code and data, and *Data space randomization* (DSR) by randomizing (encrypting) the contents of the memory. Both of them make attacks harder because the attacker either needs to find a randomized location or a key in order to modify the value of a pointer or a variable.

2.4.1 Address Space Randomization

Address space randomization (ASR) can be absolute or relative. Absolute address randomization (AAR) is coarse-grained, because it only randomizes the location of different code and data memory areas, but not the relative distances of memory objects inside these areas. The latter is only done by relative address randomization (RAR), which is more fine-grained.

Absolute address randomization

Address Space Layout Randomization (ASLR) [130] is the most prominent ASR technique, which implements AAR. It is the most comprehensive currently deployed protection against hijacking and some other attacks. In case of control-flow hijacking, if the address of the payload or gadget in the virtual memory space is not fixed, the attacker will not know what to modify the corrupted code pointer to. It also mitigates other attack types, e.g., data-only, if pointers need to be set to some particular object for a successful exploit. This is not always necessary however, e.g., a variable can simply be overwritten through a continuous overflow. In these cases ASLR, or AAR in general, does not provide any protection.

The jump target of a hijacking attack can be some injected payload in a data area or existing code in the code section. This is why every memory area must be randomized, including the stack, heap, main code segment, and libraries. The protection can always be bypassed if not *all* code and data sections are randomized. On most Linux distributions, for instance, only library code locations are randomized but the main module is at a fixed address. Most programs are not compiled as Position Independent Executables (PIE) to prevent a 10% on average performance degradation [131].

Furthermore, on some systems only 8-16 bits of randomization is used, which does not provide enough entropy to be effective against brute-force or de-randomization attacks [150]. De-randomization is often carried out by simply filling the memory with repeated copies of the payload, which is called heap-spraying or JIT-spraying [21]. Another potential attack vector is partial pointer overwrites. By overwriting the least significant byte or bytes of a pointer, it can be successfully modified to point to a nearby address [58].

A technique in the border-land between Address Space and *data space randomization* is pointer encryption. Cowan et al. [43] proposed PointGuard, which encrypts all pointers in memory and only decrypts them right before they are loaded into a register. This technique can be considered the dual of ASLR, since it also introduces entropy in addresses, but in the *data space*: it encrypts the stored address, i.e., pointers' values. To encrypt the pointers PointGuard uses the XOR operation with the same key for all pointers. Since it used only one key, by leaking out one known encrypted pointer from memory, the key can be easily recovered [156]. However the primary reason what prevented PointGuard from wide adoption was that it could break legitimate programs that cast pointers to integers.

Relative address randomization

Relative address randomization (RAR) is not widely adapted by current systems, even though there have been proposals for finer-grained randomization for both the heap and the code segment. DieHard [17] and DieHarder [124] implement RAR on the heap, by replacing the standard malloc implementation. The special allocator randomizes the allocation sites of heap objects, while ASLR only randomizes the base addresses of the entire heap. This mitigates more exploits that try to corrupt data with continuous heap overflows, but for instance it does not impact exploiting overflows inside arrays. The approach has a 20% performance overhead compared to the standard OpenBSD allocator.

Since the wide deployment of W$\oplus$X code reuse attacks (i.e., return-to-libc, ROP) became the primary threat, a lot of research has been done on code randomization [96]. To increase the entropy in code locations, researchers proposed the permutation of functions [89] and instructions inside functions [75] as well. Self-Transforming Instruction Relocation (STIR) [171] randomly re-orders the basic blocks of a binary at launch-time. While these techniques make ROP attacks harder, they usually do not protect against return-to-libc attacks. These techniques also assume that a code reuse (ROP) exploit needs several gadgets, in which case the provided entropy is high enough. However, sometimes a single gadget is enough to carry out a successful attack. The address of a single instruction, gadget, or function is relatively easy to acquire via an information leak. More recently compiler, based tools were developed as well to carry out relative address randomization, through NOP insertion. The overhead of this technique can be made negligible by taking advange of profiling information [77].

The primary attack vector against all randomization techniques is exploiting information leaks. As Figure 2.1 shows, they are always possible if (some level of) *memory safety* is not enforced. Even if everything is randomized with very high entropy (e.g., on x64 machines), information leaks can completely undermine protection, because the attacker might gain access to the secrets the defense relies on.

2.4.2 Data Space Randomization

Data space randomization (DSR) [18] was introduced by Bhatkar and Sekar to overcome the weaknesses of PointGuard and to provide stronger protection. Similar to PointGuard, DSR randomizes the *representation* of data stored in memory, not the location. It encrypts all variables, not only pointers. For a variable v, a key or mask m_v is generated. The code is instrumented to mask and unmask variables when they are stored and loaded from memory. Since several variables can be stored and loaded by the same pointer dereference, variables in equivalent "points-to" sets have to use the same key. The computation of these sets requires a static pointer analysis prior to the instrumentation. The protection is stronger, because encrypting all variables not only protects against control-flow hijacks, but also data-only exploits. Also, the use of multiple keys prevents the trivial information leak described in PointGuard's case, but not in all cases [156].

The average overhead of DSR is 15% on a custom benchmark. The solution is not binary compatible. Protected binaries will be incompatible with unmodified libraries. Also, whenever points-to analysis is needed, modularity will be an issue. Different modules cannot be handled separately, because the points-to graph has to be computed globally. Another technique called WIT also uses the points-to graph to enforce a similar policy, but deterministically. We will discuss it in detail in Section 2.5.2.

2.5 Generic Protections

First we discuss generic protection mechanisms that try to protect against most, if not all, attack types. Enforcing *memory safety* means stopping the initial memory corruption in the first two steps, and thus stopping all exploits. *Data integrity* and *data-flow integrity* are weaker policies than *memory safety*. They aim to protect against both control data (hijacking) and non-control data attacks, but not against information disclosures. While the *data integrity* prevents data corruption, *data-flow integrity* detects it.

2.5.1 Memory Safety

Enforcing *memory safety* prohibits memory corruption by *preventing* both spatial and temporal errors. Type-safe languages enforce this policy by disallowing direct pointer arithmetic, checking object bounds at array accesses, and using automatic garbage collection (instead of manual memory management). To enforce a similar policy for

C/C++, which allows pointer arithmetic and manual memory management, the objects' bounds and allocation information have to be tracked. This meta-information is either associated with the pointers or with the objects, but perfect *memory safety* can be achieved only in the former case. Therefore, in this section we only discuss pointer based approaches, that instrument pointer operations in existing unsafe code. We first cover spatial, then temporal safety.

Spatial safety

In order to enforce *spatial safety* we need to keep track of pointer bounds (the lowest and highest valid address the pointer can point to). This can be done by using *fat-pointers*, where pointers are stored together with their bounds. The original pointer representation is extended to a structure that includes the start and end address of the pointed object, along with the current value of the pointer. When the pointer is dereferenced, it is checked whether its value is still between its lower and upper bound. Doing this for all pointer dereferences however, can make program execution multiple times slower.

CCured [118] and Cyclone [83] reduced the overhead of fat-pointers by eliminating many unnecessary checks. Unfortunately, these systems need source-code annotations (e.g., to set the usage type of pointers), which made them less practical for large code bases. Furthermore, a more general problem with fat-pointers is that changing the representation of pointers changes the memory layout, which breaks binary compatibility.

CMemSafe [175] solved this problem by storing the pointer metadata separately and keeping the pointer representation unchanged. Some compatibility problems remained however, for instance it could break programs using arbitrary casts. SoftBound [116] is the latest approach in this line of work, which tries to eliminate the remaining compatibility problems and further reduce the overhead. It also keeps the bounds information decoupled from the pointers in memory. A lookup table is used to map pointers to the metadata.

The code is instrumented to propagate the metadata and to check the bounds whenever a pointer is dereferenced. For new pointers, the bounds are set to the starting and ending address of the object it is pointed to. Runtime checks at each pointer dereference ensure that the pointer stays inside bounds. These checks stop all *spatial* errors in the second step of our exploit model.

Pointer based bounds checking is capable of enforcing spatial safety completely without false positives or false negatives if and only if every module is protected. SoftBound is formally proven to provide complete spatial violation detection. Unfortunately, the performance overhead of SoftBound is high, 67% on average. While pointer based approaches, e.g., SoftBound, provide a limited compatibility with unprotected libraries, full compatibility is hard to achieve. Consider, for instance, a pointer created by the protected module. If that pointer is modified by an unprotected module, the corresponding metadata is not updated, causing false positives. We summarize the properties of the main approaches we cover at the end of the paper in Table 2.3.

Temporal safety

Spatial safety alone does not prevent all vulnerabilities, as use-after-free and double-free bugs can still be exploited. One possibility to avoid dangling pointers is to disable manual object deallocation (`free` and `delete`) in the first place, by adding a garbage collector. CCured [118], for instance, ensured *temporal safety* by disabling `free` and replacing `malloc` with the Boehm-Demers-Weiser [23] conservative garbage collector. In many application domains however, C and C++ is used because control over deallocations are necessary, and an automatic garbage collector would undermine this requirement.

To keep the control over deallocations, but ensure dangling pointers are never dereferenced, the above discussed pointer-based bounds checking approaches can be extended to check for temporal safety as well. Maintaining not only bounds but also allocation information with pointers allows enforcing full *memory safety*. Allocation information tells if the pointed-to object is still valid and has not been freed yet. It is not enough to just keep an extra bit associated with each pointer indicating the object's validity, because all pointers pointing to it have to be found and updated when the object is freed.

CMemSafe [175] uses a unique identifier (or capability) based approach to eliminate the redundancy problem of the above described naïve idea. A unique ID is associated with each allocated memory block, and the ID is stored in a global container. Each pointer is associated with the pointed block's ID and the location of the ID inside the global container. When a memory block gets freed, its ID is changed to *invalid* in the global container. Invalidated slots in the container can be reused by IDs of new allocations. This way, a pointer dereference is only valid if the associated ID and the ID in the global container match.

CETS [117] is the temporal safety extension of SoftBound. It uses the same unique identifier based approach as CMemSafe, but like SoftBound it is more compatible with programs using arbitrary casts and it has slightly reduced performance overhead. CETS is formally proven to enforce temporal safety, if spatial safety is also enforced. In other words, together with SoftBound, CETS enforces *memory safety*.

The average execution overhead of CET's instrumentation enforcing temporal safety alone is 48%. When coupled with SoftBound to enforce complete *memory safety*, the overhead is 116% on average on the SPEC CPU benchmark. As a pointer based solution, CETS suffers the same binary compatibility issues as SoftBound when it comes to unprotected libraries.

2.5.2 Data Integrity

A *data integrity* policy is an approximation of *memory safety*. Protection mechanisms of this category typically use the alternative of the previous approach, i.e., by associating meta-information with the objects instead of the pointers.

Knowing the locations and bounds of valid objects however, is not enough to decide if a pointer dereference targets the *correct* object. Therefore some object based techniques, e.g., Jones & Kelly's [84]) check pointer arithmetic instead of dereferences to check bounds. According to our model in Figure 2.1 this means the checks are done in Step 1 instead of Step 2. The checks ensure that a pointer referencing into an object stays in the object's bounds during pointer arithmetic.

One problem with this approach, however, is that pointers can legitimately go out of bounds as long as they are not dereferenced. For instance, during the last iteration of a loop over an array, a pointer typically goes off the array by one, but it is not dereferenced. Jones and Kelly (J&K), solved this problem by padding allocated objects with an extra byte. This still caused false alarms when a pointer legitimately went out of bounds more than one byte. A more generic solution to this problem was later provided by CRED [138]. J&K suffers a large performance overhead of 11-12x. CRED decreased this overhead to around 2x, but by reducing the checked data structures to character arrays only.

Dhurjati et al. [52] built on a technique called "automatic pool allocation" [99] to reduce the overhead further. Automatic pool allocation partitions the heap into pools based on a static points-to analysis and merges all target objects of a pointer into a single pool. This ensures that there is a unique pool corresponding to each pointer and each pool will only contain type homogeneous objects with a known type. When used with J&K's bounds checker, pool partitioning allows using a separate and much smaller data structures to store the bounds metadata for each pool. This can decrease the overhead further to around 120% [50].

Baggy Bounds Checking (BBC) [6] is one of the fastest object based bounds checkers. BBC trades memory for performance and adds padding to every object so that its size will be a power of two and aligns their base addresses to be the multiple of their (padded) size. This property allows a compact bounds representation and an effective way to look up object bounds. The authors of BBC claim that their solution is around twice as fast than the previously mentioned Dhurjati's automatic pool allocation based optimization. BBC's average performance overhead is 60% on the SPECINT 2000 benchmark. PAriCheck [179] was developed concurrently with BBC. It pads and aligns objects to powers of two as well for efficient bounds checking. It has slightly better performance cost and memory overhead than BBC.

Tools like Valgrind's Memcheck, Light-weight Bounds Checking (LBC) [73] and Google's AddressSanitizer (ASAN) also keep track where the objects are, but they do not check pointer arithmetic. They only mark the active object's location in a shadow memory space and ensure that dereferenced pointers point to valid objects. By leaving some space ("red-zones" or "guards") between objects they detect contiguous buffer overflows, but not indexing bugs or corruption inside objects.

The performance overhead of LBC and ASAN can often be more than 2x. Some researches proposed leveraging static pointer analysis to improve also these techniques that do not do pointer arithmetic checking. On the one hand many unnecessary checks can be eliminated statically. On the other hand, the static points-to set information could also be used to make the enforced policies stricter [178].

Write Integrity Testing (WIT) [5] calculates distinct points-to sets for every pointer dereferenced for a write and associates an ID for it. These IDs mark the objects in a shadow memory area and are checked before each (indirect) write. A drawback of enforcing different points-to sets is that the approach (similarly to the other pointer analysis based approaches like DSR) is incompatible with shared libraries. The established points-to sets depend on the whole program, which means that different programs would need e.g., different C libraries using different IDs. The only way to remain compatible is to use a single ID (for "marked"), which degenerates to the policy enforced by LBC or ASAN.

Further, since WIT does not protect reads, data, including function pointers, can be corrupted when read into a register through a corrupt pointer. To compensate this limitation, WIT applies *control-flow integrity* [1], by statically establishing and checking the targets of indirect calls too. We will discuss CFI in detail in Section 2.6.2. The reported performance overhead of WIT is between 10-25% on the SPEC benchmark.

While WIT works at compile time, BinArmor [153] aims to enforce a similar policy with binary rewriting. Since the pointer analysis the policy requires is infeasible to do in binaries, the system tries to identify potentially unsafe dereferences and their valid targets dynamically, by running and tracing the program with various inputs. This approach can neither guarantee the lack of false negatives, nor false positives, and its performance overhead can go up to 180%.

Even with pointer arithmetic checks the approaches covered in this section do not detect data corruption inside objects, thus spatial safety is not fully enforced. LBC and ASAN can neither detect when a pointer is corrupted to point to another object, only pointers trying to access invalid memory areas. None of the the *data integrity* techniques can enforce full temporal safety either. They detect accesses to currently de-allocated locations, but if the location gets reused by another object, use-after-free-s remain undetected. While these techniques can and do mitigate the exploitation of use-after-free bugs (e.g., by delaying the reuse of freed memory regions), provable temporal safety needs a pointer-based approach. ASAN avoids accidental false negatives by making sure the freed areas get reused only a long time after deallocation, but this does not stop intentional attacks.

Pool allocation based systems [52], while not enforcing temporal safety, do thwart most dangling pointer attacks due to enforcing type safe accesses on the heap. This prevents the typical use-after-free attack described in Section 2.1. Enforcing the same type safe memory reuse can also be done by replacing `malloc` with a special allocator like Cling [4]. Cling allows address space reuse only among objects of the same type and alignment. Nevertheless, none of these tools can provide the guarantees of the pointer based approaches.

2.5.3 Data-flow Integrity

Data-flow integrity (DFI) [34] detects data corruption before the data is used by checking the target of memory reads. DFI restricts reads based on the last instruction that wrote the read location. In program analysis terms, DFI enforces the reaching definition sets. The reaching definition set of an instruction is the set of instructions which might have last written (defined) the value that is used by the given instruction based on the control-flow graph.

For instance, the policy ensures that (i) the `isAdmin` variable was last written by the write instruction that the source code defines and not by some rogue attacker-controlled write or (ii) the return address used by a return was last written by the corresponding call instruction. DFI builds on static points-to analysis in order to compute the global reaching definition sets. The resulting reaching definition sets are assigned a unique ID. Each written memory location is marked in a shadow memory with the ID of the writing instruction. For each read the ID is checked to be in the set of valid IDs.

Similarly to all solutions relying on pointer analysis, independent transformation of shared libraries is an issue. DFI is not binary compatible either, since false alarms can be caused by the lack of metadata maintenance in unprotected libraries. The performance overhead of this technique can be 2-3x.

Dynamic taint analysis [141] can be considered a simplified version of DFI, which does not require static analysis. A written memory location is marked in the shadow area if the written data is derived from user input (tainted). Upon reading sensitive data (e.g., a function pointer) the mark can be checked to make sure it does not explicitly depend on user data. Taint analysis however often suffers from false positives and the performance cost can be even higher than DFI without hardware support.

2.6 Control-flow Hijack Protections

In this section we discuss protection mechanisms that aim to defend against control-flow hijacks *only*. We identified two policies that focus on this attack: *control-flow integrity* and the newly introduced *code pointer integrity*. The difference between them is that the former can *detect* the attack, while the latter can *prevent* it.

2.6.1 Return Address Integrity

Stack cookies

Using *stack cookies* (or *stack canaries*) [44] was the first proposed solution against corrupting control data on the stack. A secret value (the cookie or canary) is placed between the local variables and the saved return address (and saved base pointer). If the return address is overwritten by a buffer overflow, the cookie gets overwritten as well, which can be detected before the function returns. Current implementations in GCC and LLVM use also the improvements proposed in ProPolice[60], such as putting local arrays after other variables on the stack, so those variables are harder to corrupt as well.

The protection provided by stack cookies has several weaknesses. First, it relies on the secrecy of the random canary value, which might be acquired via an information

General opt. Cookie opt.	-O2 Optimized	-O2 All func.	-O0 Optimized	-O0 All func.
LLVM	0.33%	2.38%	0.02%	4.92%
GCC	0.14%	2.77%	0.30%	3.22%

Table 2.2: Performance overhead of stack cookies on the SPEC 2006 benchmark

leak. Almost all memory corruption bugs can be exploited to leak out memory contents. The simplest example is when a buffer overrun happens on the stack while reading. In that case the memory contents past the end of the buffer might be included in the program's output as well. By corrupting other data pointers pointing to data that the program will output arbitrary memory contents can be leaked out. Format string bugs are also often exploited to leak memory contents, where a corrupted pointer can simply be created by providing a carefully crafted format string.

Second, cookies only detect corruption due to continuous buffer overflows. Overwriting the return address directly, by exploiting a different memory corruption bug (e.g., incorrect index computation), remains undetected. Of course, this attack needs the actual location of the stack, which is usually unknown when address space randomization (ASLR) is active on the system. Notice that with the above described information leaks, the current location of the stack can be easily acquired by leaking any pointer to a local variable.

We measured the performance cost of stack cookies implemented in LLVM and also in GCC using the SPEC 2006 benchmark. We ran the test with and without general optimization, and with and without stack cookie optimization. Stack cookie optimization means that cookies are not used in all functions, but heuristics are used to decide which functions should be protected (e.g., only the ones that contain local buffers).

The above table shows the average results of the eight tests. When both general and cookie optimization was on, the overhead was negligible: less then 1%. Another reason why stack cookies are popular and widely deployed besides low overhead is that they do not introduce compatibility issues.

Shadow stacks

To thwart the attacks against stack cookies, the use of *shadow stacks* was proposed [167, 40]. The shadow stack is a separated stack maintained to contain only copies of saved return addresses (and saved base pointers). At function calls the return address is pushed to the shadow stack as well, then before the function returns the original value is compared to the copy on the shadow stack.

This solution makes any (direct or continuous) corruption much harder, even when the shadow stack itself is not protected, since the attacker has to corrupt the return address at two separate locations. The reported performance overhead in previous papers using shadow stack varies between 5-40% [40, 2]. In order to get an up to date estimate for the overhead we implemented our prototype as an LLVM compiler pass.

To make our implementation not only fast but also more secure, we keep the shadow stack pointer in a dedicated register (i.e., `%r15`). This not only minimizes the cost of accessing the shadow stack, but allows us to effectively protect it via randomization (i.e., ASLR), because we can prevent information leaks. The shadow stack pointer is only stored in its dedicated register and no other pointers point to the shadow region in the original program. This means that there are no pointers to the shadows stack in the memory, therefore there is nothing to leak out.

Some unprotected low-level libraries however might spill the register contents. One case we identified when the shadow stack pointer can spill to memory is at a `setjmp`, when all registers are saved into the `setjmp` buffer. To prevent this we simply modified `setjmp` to not save this register. The shadow stack pointer does not need to be saved in case of a `longjmp`, because we can pop off return addresses from the shadow stack until one matches.

The average performance overhead of our solutions measured on the SPEC 2006 benchmark was 4.8%, which is comparable to stack cookies when they are applied to all functions. However, using a dedicated register as the shadow stack pointer we gave up compatibility with unmodified libraries. Libraries are not required to use the shadow stack, but they have to be recompiled so that they do not touch the dedicated register. Implementation not relying on a dedicated register on the other hand can have higher overheads. Dang et al. [45] made a more detailed comparison of shadow stack performance overheads.

The primary reason shadow stacks are not deployed are still compatibility issues. For instance some low-level libraries might use return instructions not only to return from functions, but e.g., to jump to the landing pad of an exception [133]. Shadows stack implementation need to be made compatible with such low-level functions.

2.6.2 Control-flow Integrity

Control-flow integrity (CFI) techniques protect *all* control-flow transfers, including indirect calls and jumps, not only returns. They are similar to Return Address Integrity techniques in the sense that they do not prevent the malicious modification of function pointers or return addresses; instead some checks are carried out before the control transfer to detect corruption. They differ in that CFI techniques rely on prior knowledge of all valid targets for each control-flow transfer. For instance, static pointer analysis can be used to establish the valid targets (points-to sets) of indirect function calls and returns. By assigning different IDs to distinct points-to sets, we can mark the source and target locations of indirect control transfers. We can detect pointer corruption, by checking if source and destination IDs match, right before using the destination pointer at the control transfer instruction.

CFI was originally introduced by Abadi et al. [1]. They proposed placing the IDs inside the code itself to protect them through *code integrity*. The IDs are encoded into instructions to not affect the semantics of the code. All indirect calls and returns are instrumented to check the target address if it has the correct ID before jumping there. This mechanisms relies on *non-executable data* to prevent forging of valid targets by simply placing an ID before an injected shellcode. The technique however did not get adopted in practice for a long time, presumably because of two reasons. One is that without source code or debug information it was not clear how to establish the valid targets of indirect control transfers. The other reason is that the reported performance overhead could be as high as 46%, which is usually considered too high for a practical protection.

We proposed CCFIR [185] to solve these issues. It was the first CFI solution that demonstrated the feasibility of protecting commercial off-the-shelf Windows binaries, while providing excellent performance. The insight that allowed us to transform stripped binaries was that, with the wide deployment of ASLR, Windows executables contain enough information in relocation tables which can be used to find all legal instructions and jump targets reliably. Further, we designed CCFIR so that the jump target validity checks are as fast as possible. To achieve this, for each valid control transfer target we create a jump stub in the transformed program and place these stubs at special memory locations in the virtual memory space. Doing this allows us to determine the validity of a target just based on its address, simply testing whether the right bits are set in the target pointer value. With this design we managed to reduce the overhead to 3.6%/8.6% (average/max). CCFIR also enabled incremental deployment by supporting unmodified libraries, and added an extra layer of protection through randomization.

BinCFI [187] was another practical CFI implementation, but for Linux binaries. More recently compile time CFI solutions have appeared as well [162]. CFI in the LLVM compiler, for instance, can enforce that virtual function call targets in C++ belong to the appropriate classes or that other indirectly called functions have the correct type signature.

The strength of any CFI protection has hard limits. First of all, enforcing even the strictest CFI policy that is theoretically possible is not enough to stop all possible control-flow hijacks. If a function pointer can point to function f and g, an attacker will always be able to switch these targets up. Consider for example that f implements the “always allow access” and g implements the “check permissions” function. For instance, the Linux Security Modules architecture allows the implementation of different kinds of access control (e.g., SELinux) checks through similar function pointer hooks.

In practice, even unrelated targets can be switched up because of multiple reason. First, due to the conservativeness of any pointer analysis, the resulting points-to sets are always over-approximations. Second, in order to have unique IDs, each points-to set which includes a common target must be merged. Furthermore, all exported functions in shared libraries must be marked with the same ID, because of potential (external) aliasing.

This is why all practical implementations actually enforce a much weaker policy, which restricts indirect control transfers to the union of all their points-to sets. This typically means that all indirectly callable functions are in a single set of allowed call targets, and all call sites are in a single set of allowed return targets. The advantage of this policy is that it does not need pointer analysis; it suffices to enumerate all functions whose addresses are taken. Both the original CFI implementation and the covered more practical implementations can enforce only this coarse-grained policy. It has been shown by many researchers [69, 46, 33] that this provides insufficient protection and all of these techniques can be bypassed in a principled way.

2.7 Comparative Analysis of Protection Mechanisms

We summarize the properties of a selected set of solutions in Table 2.3, grouped by the policy categories identified in Section 2.1. We do not cover *instruction set randomization*, because the same attack vectors can be defeated with page permission enforcing *code integrity* and *non-executable data*. We will discuss *code pointer integrity* separately in Chapter 3. For the rest of the policies we include one or a few of the protection mechanisms that belong to that category. Most of the techniques are compiler based, but we include some static binary rewriting based one as well. We omit dynamic binary instrumentation solutions due to their high performance cost, or others which are not fully automatic (e.g., need source code modification). The upper half of the table covers protections aiming to protect against memory corruption in general and thus mitigate all four different attacks identified in Section 2.1. The lower half covers approaches which aim to primarily protect against control-flow hijacks only.

The performance is represented as the average and maximum overhead using either the SPEC CPU 2000 or 2006 benchmarks. We rely on the numbers reported by the developers of the tools, since several of them are not publicly available. We stress that since the values represent measurements in different environments, different configurations, and sometimes with different sets of programs, they only provide rough estimates.

Techniques with stronger protections tend to have higher overhead. Enforcing full *memory safety* typically makes execution two or more times slower. *Data integrity* techniques trade off their protection strengths for lower overhead, however tools such as ASAN are still only used for debugging purposes. Many CFI techniques have acceptable overhead even for production code however.

	Policy type (main approach)	Technique	Performance % avg.	max.	Dep.	Compatibility	Primary attack vectors
Generic prot.	Memory Safety	SofBound + CETS	116	300	×	Binary	—
	Data Integrity	BBC	60	127	×	—	UAF, sub-obj
		LBC, ASAN	23	175	×	—	UAF, sub-obj, read corruption
		WIT	10	25	×	Binary/Modularity	UAF, sub-obj, read corruption
	Data Space Randomization	DSR	15	30	×	Binary/Modularity	Information leak
	Data-flow Integrity	DFI	104	155	×	Binary/Modularity	Approximation
CF-Hijack prot.	Code Integrity	Page permissions (R)	0	0	✓	JIT compilation	Code reuse or code injection
	Non-executable Data	Page permissions (X)	0	0	✓	JIT compilation	Code reuse
	Address Space Randomization (AAR)	ASLR	0	0	✓	Relocatable code	Information leak
		ASLR (PIE on 32 bit)	10	26	×	Relocatable code	Information leak
	Address Space Randomization (RAR)	NOP insertion	2	25	×	—	Information leak
	Return Address Integrity	Stack cookies	0	5	✓	—	Direct overwrite
		Shadow stack	5	12	×	Exceptions	Corrupt function pointer
	Control-flow Integrity	Abadi CFI	8	56	×	Binary/Modularity	Approximation
		WIT (CFI part)	10	25	×	Binary/Modularity	Approximation
		CCFIR	4	9	×	—	Approximation
		BinCFI	5	45	×	—	Approximation

Table 2.3: This table groups the different protection techniques according to their policy and compares the performance impact (measured on SPEC benchmarks), large-scale deployment status (dep.), compatibility issues, and main attack vectors that circumvent the protection. "UAF" stands for use-after-free and "sub-obj" for corruption inside objects.

The more serious issue for many solution, that also prevents wide adoption, is compatibility. All solutions relying on static pointer analysis, such as WIT, DSR, and DFI, have problems supporting shared libraries. On the one hand, they are binary incompatibility with unmodified shared libraries which do not maintain the metadata used and required by the transformed code. On the other hand, they lack modularity, because they need access to the source code of the whole program, including libraries, in order to work correctly. Even when this is possible, the over-approximation of the points-to sets established by (the already conservative) static analysis of these tools go to the extreme. Think of WIT using only a single ID, or DSR a single key. These are the reason why existing CFI solutions do not even use pointer analysis, but establish allowed targets based properties, such as the address of the function was taken or its type signatures.

Regarding the strength of the protections, none of them can stop all attacks, except the ones enforcing complete *memory safety* with pointer based techniques. Protections enforcing *data integrity* policies have more attack vectors, like use-after-free (UAF), corruption of sub-objects (sub-obj) or corrupting values in registers via read dereferences. Randomization techniques provide the most comprehensive solutions as a generic and hijacking protection respectively, but all of them can be circumvented by information leaks. The protection level of DFI is only bounded by the approximation, due to the conservativeness of static analysis. This approximation applies to all other static pointer analysis based approach.

To summarize, the deployed protections are too weak, enforcing memory safety is too expensive, while the solutions enforcing policies between these extremes either have serious compatibility issues and/or have too high overhead to be incorporated into widely deployed software. Considering control-flow hijacking only, *control-flow integrity* solutions do have acceptable overhead, but in order to avoid compatibility issues they weakened their policies so much that their protection can be circumvented. We introduced a new policy, called *code pointer integrity*, which would fully prevent all such attacks, as opposed to CFI, which can only detect some of them. In the next chapter we describe the technique to enforce this policy, which meets our performance, compatibility and protection strength requirements we set up in this chapter.

Chapter 3

Code-pointer Integrity

As the previous chapter has shown, none of the deployed defense mechanisms (such as ASLR or DEP) are able to stop control-flow hijacking attacks. Protections enforcing control-flow integrity have limited guarantees [69, 46, 33], and enforcing memory safety is too costly. This chapter describes how to enforce *Code-pointer Integrity* (CPI), a new low-level security policy introduced in the previous chapter. The CPI policy guarantees the integrity of all code pointers in a program (e.g., function pointers, return addresses) and thereby prevents *all* control-flow hijack attacks.

We also introduce *Code-pointer Separation* (CPS), a relaxation of CPI with better performance properties, that enforces a weaker policy than CPI. Both CPI and CPS offer substantially better security-to-overhead ratios than the state of the art. They are practical (we protect a complete FreeBSD base system and packages like `apache` and `postgresql`), effective (prevent all attacks in the RIPE benchmark), and efficient: on SPEC CPU2006, CPS averages 1.2% overhead for C and 1.9% for C/C++, while CPI's overhead is 2.9% for C and 8.4% for C/C++.

Both CPI and CPS rely on *SafeStack*, which can also be used separately as a better return address protection than stack cookies or a shadow stack. SafeStack is already part of the LLVM/Clang compiler.

3.1 Threat Model

This chapter is concerned solely with control-flow hijack attacks, namely ones that give the attacker control of the instruction pointer as described in Section 2.1. Recall that the purpose of this type of attack is to divert control flow to a location that would not otherwise be reachable in that same context, had the program not been compromised. Examples of such attacks include forcing a program to jump (i) to a location where the attacker injected shell code, (ii) to the start of a chain of return-oriented program fragments ("gadgets"), or (iii) to a function that performs an undesirable action in the given context, such as calling `system()` with attacker-supplied arguments. Data-only attacks and information disclosures are out of scope.

We assume powerful yet realistic attacker capabilities: full control over process memory, but no ability to modify the code segment. Attackers can carry out arbitrary memory reads and writes by exploiting input-controlled memory corruption errors in the program. They cannot modify the code segment, because the corresponding pages are marked read only, they can not control the program loading process either. Our assumptions are consistent with prior work in this area.

3.2 Design

The *code-pointer integrity* policy prohibits the corruption of code pointers through memory corruption bugs. This means that neither spatial nor temporal errors can be exploited to maliciously modify any code pointer during program execution. We now present the mechanism to enforce this policy. We protect function pointers and return addresses differently. CPI and CPS only differ in the function pointer protection and they both use our SafeStack mechanism to protect return addresses. The difference between CPI and CPS is that the CPS mechanism trades off some security guarantees for performance. We first describe the design of the CPI function pointer protection, then SafeStack, and finally, CPS.

3.2.1 Enforcing the Integrity of Function Pointers

In order to enforce the integrity of code pointers, we need to prevent their corruption through memory corruption exploits. This means that we need to prevent code pointer access through invalid pointer dereferences. An invalid pointer, as defined in Section 2.1.1, is an out of bounds or dangling pointer. Note that when *memory safety* is enforced, no such dereference is allowed, so *code-pointer integrity* is ensured.

To enforce precise *memory safety*, the bounds and temporal information of pointers need to be tracked [116, 118, 83], as described in Section 2.5.1. Unfortunately, this has a high runtime overhead. SoftBounds+CETS [116, 117], the most advanced implementation of pointer metadata tracking, slows down the execution by 2×. Our observation is that only a small subset of pointers are responsible for making control-flow transfers, and we can enforce *code-pointer integrity* by enforcing memory safety only for the control-sensitive pointers.

Sensitive pointers are code pointers and pointers that may later be used to access sensitive pointers. Put another way, the set of sensitive pointers is the transitive closure of the “pointed-by” relation on the set of code pointers. This recursive definition is illustrated in Figure 3.1. The precise set of sensitive pointers changes dynamically at run time. For example, pointer 2 in Figure 3.1 with type `void*` is sensitive when it points to another sensitive pointer, but it is not sensitive when it points to an integer. However, the CPI policy can still be enforced using any over-approximation of this set. We obtain such over-approximations at compile time, using static analysis.

Once we established the set of sensitive pointers, we ensure at runtime that code

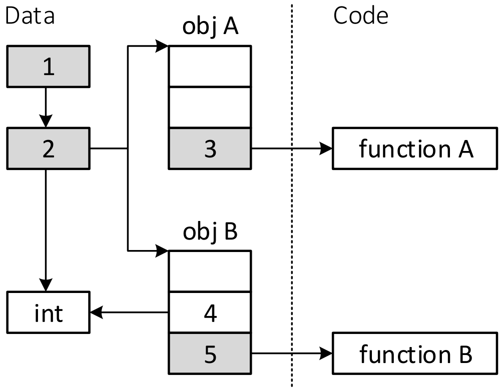


Figure 3.1: CPI protects code pointers 3 and 5 and data pointers 1 and 2 (which may access pointers 3 and 5 indirectly). Pointer 2 of type `void*` may point to different objects at different times. The `int*` pointer 4 and non-pointer data locations are not protected.

pointers (and other sensitive pointers) can only be accessed in a memory safe way. To this end, we transform the original program to store each sensitive pointer and its metadata in a separate, safe memory region that is inaccessible through other pointers. Then we instrument all sensitive pointer operations to keep track of their bounds and temporal metadata and use it to check if dereferences are safe. Recall that we use this technique for protecting function pointers. We will use our SafeStack mechanism to protect return addresses. We now proceed to describe the CPI static analysis and instrumentation in more detail.

CPI static analysis

We determine the set of sensitive pointers using type-based static analysis: a pointer is sensitive if its type is sensitive. Sensitive types are: pointers to functions, pointers to sensitive types, pointers to composite types (such as `struct` or array) that contains one or more members of sensitive types, or universal pointers (i.e., `void*`, `char*` and opaque pointers to forward-declared `struct`s or `class`es). A programmer could additionally indicate, if desired, other types to be considered sensitive, such as `struct ucred` used in the FreeBSD kernel to store process UIDs and jail information. All code pointers that a compiler or runtime creates implicitly (such as return addresses, C++ virtual table pointers, and `setjmp` buffers) are sensitive as well.

Our static analysis finds all program instructions that manipulate sensitive pointers. These instructions include pointer dereferences, pointer arithmetic, and memory allocation and deallocation operations (`new`/`delete`, `malloc()`/`free()`/etc.).

The derived set of sensitive pointers is an over-approximation: it may include universal pointers that never end up pointing to sensitive values at runtime. For instance, the C/C++ standard allows `char*` pointers to point to objects of any type, but such pointers are also used for C strings. As a heuristic, we assume that `char*` pointers that are passed to the standard `libc` string manipulation functions or that are assigned to point to string constants are not universal. Neither the over-approximation nor the `char*` heuristic affect the security guarantees provided by CPI: over-approximations merely introduce extra overhead, while heuristic errors may result in false violation reports (though we never observed any in practice).

Memory region manipulation functions from `libc`, such as `memset` or `memcpy`, could introduce a lot of overhead. Since these functions take `void*` arguments, their instrumented version propagates the metadata for the whole region, regardless of whether they contain sensitive data or not. CPI's static analysis instead detects cases when the region is not sensitive, by analyzing the real types of the arguments prior to being cast to `void*`. In these cases the uninstrumented version of these functions are used.

We augmented type-based static analysis with a data-flow analysis that handles most practical cases of unsafe pointer casts and casts between pointers and integers. If a value v is ever cast to a sensitive pointer type within the function being analyzed, or is passed as an argument or returned to another function where it is cast to a sensitive pointer, the analysis considers v to be sensitive as well. This analysis may fail when the data flow between v and its cast to a sensitive pointer type cannot be fully recovered statically, which might cause false violation reports. Such casts are a common problem for all pointer-based memory safety mechanisms for C/C++ that do not require source code modifications [116].

A key benefit of CPI is its selectivity: the number of pointer operations deemed to be sensitive is a small fraction of all pointer operations in a program. As we show in Section 3.4, for SPEC CPU2006, the CPI type-based analysis identifies for instrumentation 6.5% of all pointer accesses; this translates into a reduction of performance overhead of $16 - 44\times$ relative to full memory safety.

Nevertheless, we still think CPI can benefit from more sophisticated analyses. CPI can leverage any kind of *points-to* static analysis, as long as it provides an over-approximate set of sensitive pointers. For instance, when extending CPI to also protect select non-code-pointer data, we think DSA [97, 100] could prove more effective.

CPI Instrumentation

Our instrumentation directly follows the design of SoftBounds+CETS [117], but works only on sensitive pointers. We also create an isolated new safe memory region (Figure 3.2) to store the metadata in. Note that SoftBounds does not need such separate isolation primitive, as the enforced spatial memory safety already protects its metadata store. Another difference is that CPI *also* stores the value of the protected pointers together with their metadata, in the so called *safe pointer store*.

We instrument the program in order to (i) ensure that all sensitive pointers are stored in the *safe pointer store*, (ii) create and propagate metadata for such pointers at runtime, and (iii) check the metadata on dereferences of such pointers. Storage

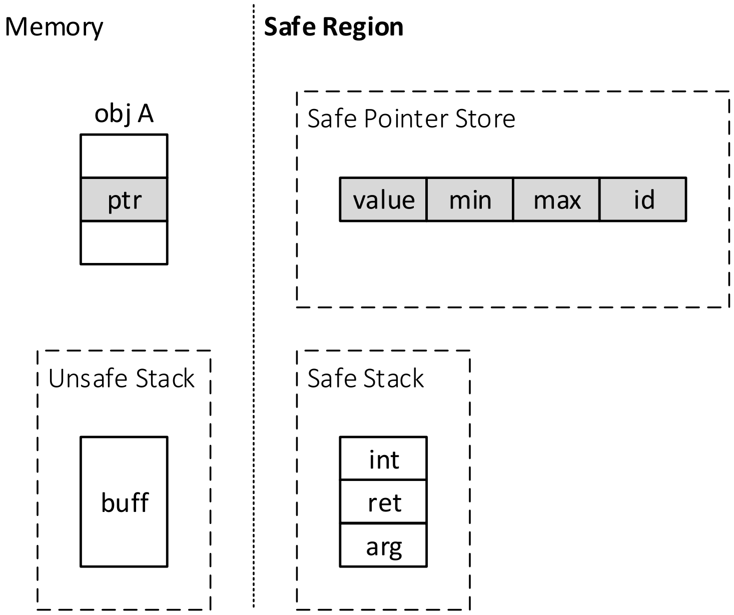


Figure 3.2: CPI memory layout: The safe region contains the safe pointer store and the safe stack (or stacks). The location of a sensitive pointer on the left (shaded) remains unused, while the value of this pointer and its metadata are stored in the safe pointer store. Each safe stack has a corresponding unsafe stack outside the safe memory region, to allocate any unportected objects.

space for sensitive pointers is allocated in both the safe region (the safe pointer store) and the regular region (as usual); one of the two copies always remains unused. This is necessary for universal pointers (e.g., `void*`), which could be stored in either region depending on whether they are sensitive at run time or not, and also helps to avoid some compatibility issues that arise from the change in memory layout.

The address in regular memory is used as an offset to look up the value of a sensitive pointer in the *safe pointer store*. The safe pointer store maps the address `&p` of sensitive pointer `p`, to the value of `p` and its associated metadata. The metadata for `p` contains the lower and upper address bounds of the pointed object, and the temporal ID as in case of SoftBounds+CETS (see Figure 3.2). When combined with the isolation of the safe region (Section 3.2.2), this allows CPI to guarantee full memory safety of all sensitive pointers without having to instrument all pointer operations.

The instrumentation changes instructions that operate on sensitive pointers, as found by CPI's static analysis, to create and propagate the metadata. We initialize a new metadata entry at instructions that explicitly take the address of a statically allocated memory object or a function, allocate a new object on the heap, or take an address of a sub-object. Instructions that compute pointer expressions are instrumented to propagate the metadata accordingly. Instructions that load or store sensitive pointers to memory are replaced with CPI intrinsic instructions (Section 3.2.2) that load or store both the pointer values and their metadata from/to the safe pointer store. In principle, call and return instructions also store and load code pointers, and so would need to be instrumented, but we instead protect return addresses using a *safe stack* (Section 3.2.3).

Every dereference of a sensitive pointer is instrumented to check at runtime whether it is safe, using the metadata associated with the pointer being dereferenced. Together with the restricted access to the safe region, this results in precise memory safety for all sensitive pointers.

Universal pointers (`void*` and `char*`) are stored in either the safe pointer store or the regular region, depending on whether they are sensitive at runtime or not. CPI instruments instructions that cast from non-sensitive to universal pointer types to assign special "invalid" metadata (e.g., with lower bound greater than the upper bound) for the resulting universal pointers. These pointers, as a result, would never be allowed to access the safe region. CPI intrinsics for universal pointers would only store a pointer in the *safe pointer store* if it had valid metadata, and only load it from the safe pointer store if it contained valid metadata for that pointer; otherwise, they would store/load from the regular region.

CPI can be configured to simultaneously store protected pointers in both the safe pointer store and regular regions, and check whether they match when loading them. In this debug mode, CPI detects all *attempts* to hijack control flow using non-protected pointer errors; in the default mode, such attempts are silently prevented. The silent mode has a smaller overhead as it only stores the pointer value once and no comparisons are made. Second, the execution might be able to continue in case of an attack attempt. The debug mode provides better compatibility with non-instrumented code, e.g., shared libraries, that may read protected pointers (for example, callback addresses) but not write them. Note that compatibility with uninstrumented shared libraries is not guaranteed in either case. If the protected program uses pointers created by an uninstrumented library, the metadata will be missing.

The CPI instrumentation pass precedes compiler optimizations, thus allowing them to potentially optimize away some of the inserted checks while preserving the security guarantees.

3.2.2 Isolating the Safe Region

The safe region can only be accessed via CPI intrinsic instructions, and they properly handle pointer metadata and the safe stack (Section 3.2.3). The mechanism for achieving this isolation is architecture-dependent.

On x86-32, we rely on hardware segment protection. We make the safe region accessible through a dedicated segment register, which is otherwise unused, and configure limits for all other segment registers to make the region inaccessible through them. The CPI intrinsics are then turned into code that uses the dedicated register and ensures that no other instructions in the program use that register. The segment registers are configured by the program loader, whose integrity we assume in our threat model; we also prevent the program from reconfiguring the segment registers via system calls.

On x86-64, CPI can rely on information leak free randomization, or software fault isolation (SFI) [35]. The randomization based protection relies on the fact that no addresses pointing into the safe region are ever stored in the regular region. This isolation primitive no longer enforces the segment limits, however it still provides two segment registers with configurable base addresses. Similarly to x86-32, we use one of these registers to point to the safe region, however, we choose the base address of the safe region at random and rely on preventing access to it through information hiding. Unlike classic ASLR though, our hiding is leak-proof. Since the objects in the safe region are indexed by addresses allocated for them in the regular region, no addresses pointing into the safe region are ever stored in regular memory at any time during execution. The 48-bit address space of modern x86-64 CPUs makes guessing the safe region address impractical, because most failed guessing attempts would crash the program, and such frequent crashes can easily be detected by other means.

The SFI based isolation primitive restricts the addressable memory with an upper bound by instrumenting all memory writes. We align the safe region in memory so that with a single bitmask operation on pointers we can restrict access to it. An important aspect is that we do not need to instrument accesses to the safe stack, as they are guaranteed to be safe. This makes our instrumentation cheap, introducing an additional 5% overhead relative to hardware-enforced segmentation. While randomization-based protection is probabilistic, and might be susceptible to other types of information leaks or brute force attacks, SFI protection gives deterministic guarantees.

Since sensitive pointers form a small fraction of all data stored in memory, the safe pointer store is highly sparse. To save memory, it can be organized as a hash table or a two-level lookup table. The two-level lookup table is the same data structure that the Intel MPX [82] design uses. While we implemented it in software, in the future we plan to leverage the MPX instructions to accelerate our *safe pointer store* access and also metadata check. We discuss this further in Section 3.3.4.

3.2.3 SafeStack: Enforcing the Integrity of Return Addresses

CPI enforces the integrity of return addresses using the SafeStack mechanism. A key observation is that most stack objects, including saved return addresses, are only accessed directly through the stack pointer with a constant offset. If these objects cannot be accessed via other pointer dereferences, their integrity can be proved safe statically, without any run time checks and metadata. We therefore create two stacks, one for safely accessed objects only and one for the rest. We call the two stacks the *safe stack* and the *unsafe stack*. We keep the *safe stack* inside the safe region and thus forbid any access to it through unsafe pointer dereferences. Note that this way not only return addresses are protected, but all scalar variables that are accessed exclusively within the corresponding function, without their address taken.

The idea of using a separate stack for array type and structures whose address are taken was first proposed by Bhatkar et al. [19] for hardening and was also used by XFI [59] for software fault isolation. SafeStack is the first implementation of this idea in a modern compiler, that operates on the intermediate representation level, supporting both C and C++ code. Note, that maintaining two stack is different from the *heapification* used by Cyclone [83] or CCured [118], which allocate unsafe objects on the heap, which can lead to decreased performance. The SafeStack mechanism is also different from shadow stacks, which make a copy of saved return addresses on an additional stack and thus pay a constant overhead for every function call [40, 47]. We do the inverse, and instead move the unsafe objects to a separate *unsafe stack*. An unsafe object is any object that is accessed through pointers, like arrays or variables whose address is passed to other functions. The *unsafe stack* is allocated in the regular region and only functions with unsafe objects create stack frames on it. In our experience, less than 25% of functions need such additional stack frames (see Table 3.2). Furthermore, this fraction is much smaller among short functions, for which the overhead of setting up the extra stack frame is non-negligible.

The SafeStack mechanism consists of a static analysis pass, an instrumentation pass, and runtime support. The analysis pass identifies, for every function, which objects in its stack frame are guaranteed to be accessed safely and can thus be placed on the *safe stack*; return addresses and spilled registers always satisfy this criterion. For the objects that do not satisfy this criterion, the instrumentation pass inserts code that allocates a stack frame for these objects on the regular stack. The runtime support allocates regular stacks for each thread and can be implemented either as part of the threading library, as we did on FreeBSD, or by intercepting thread create/destroy, as we did on Linux. Exceptions and longjumps are also supported, by restoring the *unsafe stack* pointer after they happen. CPI stores the *unsafe stack* pointer inside the thread control block, which is pointed to by one of the segment registers and can thus be accessed with a single memory read or write. The original stack pointer is used as the *safe stack* pointer.

Compared to a shadow stack that makes copies of saved return addresses, SafeStack presents several advantages. First, all return instruction pointers and most local variables are protected, whereas a shadow stack only protects return instruction pointers. Second, SafeStack is compatible with uninstrumented code that uses just the regular stack, and it directly supports exceptions, tail calls, and signal handlers. Third, SafeStack has near-zero performance overhead (Section 3.4.2), because only a handful of functions require extra stack frames, while a shadow stack allocates a shadow frame for every function call.

The SafeStack is now part of the LLVM/Clang compiler, and can be employed independently from CPI. By providing precise protection of all return addresses (which are the target of ROP attacks today), spilled registers, and some local variables, SafeStack provides substantially stronger security than stack cookies [44], while incurring equal or lower performance overhead and deployment complexity. Therefore, we believe it can replace stack cookies as the default return address protection mechanism in modern compilers.

3.2.4 Code-pointer Separation (CPS)

The *code-pointer separation* policy trades off some of CPI's security guarantees for reduced runtime overhead. This is particularly relevant to C++ programs with many virtual functions, where the fraction of sensitive pointers instrumented by CPI can become high, since every pointer to an object that contains virtual functions is sensitive. We found that, on average, CPS reduces overhead by 4.3× (from 8.4% for CPI down to 1.9% for CPS), and in some cases by as much as an order of magnitude.

CPS restricts the set of sensitive pointers to code pointers only, leaving (data) pointers that point to code pointers uninstrumented. Since only code pointers are instrumented, no bounds or temporal information is stored (or checked), only the code pointer itself is in the *safe pointer store*. This prevents attackers from *forging* a code pointer, e.g., from a value in the regular area, but still allows them to trick the program to interchange one valid code pointer with another, as long as are they are in the *safe pointer store* at the time. This is because with CPS, data pointers pointing to code pointers can be corrupted, so a pointer pointing to a code pointer can be maliciously changed to point to another *existing and valid* code pointer, but trying to read or write an *invalid* code pointer location will fail.

CPS is enforced similarly to CPI, except for the criteria used to identify sensitive pointers during static analysis, and that CPS does not need any metadata. Control-flow destinations (pointed to by code pointers) do not have bounds, because the pointer value must always match the destination exactly, hence no need for bounds metadata. Furthermore, they are typically static, hence do not need temporal metadata either (there are a few rare exceptions, like unloading a shared library, which are handled separately). This reduces the size of the safe region and the number of memory accesses when loading or storing code pointers. A CPS-instrumented program performs almost the same number of memory accesses when loading or storing code pointers as a non-instrumented one, as the only difference is that the pointers are being loaded or stored from the *safe pointer store* instead of their original location. As a result, CPS can be enforced with low performance overhead.

CPS guarantees that (i) code pointers can only be stored to or modified in memory by code pointer store instructions, and (ii) code pointers can only be loaded by code pointer load instructions from memory locations to which previously a code pointer store instruction stored a value. This limits the set of locations to which the control can be redirected, in particular to the set of functions whose addresses were explicitly taken by the program at run time. Combined with SafeStack, CPS precisely protects return addresses.

CPS enforces a stronger policy than most CFI implementations [1, 187, 185]. As discussed in Section 2.6.2, practical CFI implementations allow indirect calls to any function, whose address is taken (as established statically); and allow returns to any call sites. In contrast, CPS allows returns only to their single actual caller (due to SafeStack), and indirect calls to the set of function whose address has been taken during the execution up to that time, and the taken code pointer is still assigned to a variable. This dynamic, run-time set of assigned function pointers is always a subset of the statically established set of function pointers whose address is taken. Consider the following simple example:

```
void (*fptr)();
if (...) fptr = f; else fptr = g;
```

CFI would permit a call through `fptr` to go to either `f` or `g`, because CFI statically approximates the set of legitimate control-flow targets. CPS however would only permit a call to the function that was actually assigned.

To illustrate the same with a real-world example, consider the case of the Perl interpreter, which implements its opcode dispatch by representing internally a Perl program as a sequence of function pointers to opcode handlers and then calling in its main execution loop these function pointers one by one. CFI statically approximates the set of legitimate control-flow targets, which in this case would include all possible Perl opcodes. CPS however permits only calls through function pointers that are actually assigned. This means that a memory bug in a CFI-protected Perl interpreter may permit an attacker to divert control flow and execute any Perl opcode, whereas in a CPS-protected Perl interpreter the attacker could at most execute an opcode that exists in the running Perl program.

CPS provides strong control-flow integrity guarantees and incurs low overhead (Section 3.4). We found that it prevents all recent attacks designed to bypass CFI [69, 46, 33]. We consider CPS to be a solid alternative to CPI in those cases when CPI's (already low) overhead seems too high.

3.3 Implementation

We implemented a CPI/CPS enforcement tool for C/C++, called Levee, on top of the LLVM 3.3 compiler infrastructure [98], with modifications to LLVM libraries, the `clang` compiler, and the `compiler-rt` runtime. To use Levee, one just needs to pass additional flags to the compiler to enable CPI (`-fcpi`), CPS (`-fcps`), or SafeStack protection (`-fsafestack`). Levee works on unmodified programs and supports Linux, FreeBSD, and Mac OS X in both 32-bit and 64-bit modes.

3.3.1 Pointer Analysis and Instrumentation Passes

We implemented the static analysis and instrumentation for CPI as two LLVM passes, directly following the design described in Section 3.2.1. The LLVM passes operate on the LLVM intermediate representation (IR), which is a low-level strongly-typed language-independent program representation tailored for optimization purposes. The LLVM IR is generated from the C/C++ source code by `clang`, which preserves most of the type information that is required by our analysis, with a few corner cases. For example, in certain cases, `clang` does not preserve the original types of pointers that are cast to void* when passing them as an argument to memset or similar functions, which is required for the memset-related optimizations discussed in Section 3.2.1. The IR also does not distinguish between void* and char* (represents both as i8*), but this information is required for our string pointers detection heuristic. We augmented clang to always preserve such type information as LLVM metadata.

3.3.2 SafeStack Instrumentation Pass

The SafeStack instrumentation targets functions that contain on-stack memory objects that cannot be put on the *safe stack*. For such functions, it allocates a stack frame on the *unsafe stack* and relocates the corresponding variables to that frame.

Given that most of the functions do not need an *unsafe stack*, Levee uses the usual stack pointer (rsp register on x86-64) as the *safe stack* pointer, and stores the *unsafe stack* pointer in the thread control block, which is accessible directly through one of the segment registers. When needed, the *unsafe stack* pointer is loaded into an IR local value, and Levee relies on the LLVM register allocator to pick the register for the *unsafe stack* pointer. Levee explicitly encodes *unsafe stack* operations as IR instructions that manipulate an *unsafe stack* pointer; it leaves all operations that use a *safe stack* intact, letting the LLVM code generator manage them. Levee performs these changes as a last step before code generation (directly replacing LLVM's stack-cookie protection pass), thus ensuring that it operates on the final stack layout.

Certain low-level functions modify the stack pointer directly. These functions include setjmp/longjmp and exception handling functions (which store/load the stack pointer), and thread create/destroy functions, which allocate/free stacks for threads. On FreeBSD we provide full-system CPI, so we directly modified these functions to support the dual stacks. On Linux, our instrumentation pass finds setjmp/longjmp and exception handling functions in the program and inserts required instrumentation at their call sites, while thread create/destroy functions are intercepted and handled by the Levee runtime.

On x86-32, where we use segmentation for isolation, we make the stack segment and the data segment non-overlapping, so the *safe stack* is only accessible through the segment register. When using randomization for isolation, it is important that the (safe) stack pointer does not leak. Some low level libraries, including libc, can leak the stack pointer in several ways, such as during a longjmp, signal handling, user-level context switching related functions, etc. Note, that dumping the stack pointer on the safe stack is not a problem, but dumping it elsewhere might expose it to leakage. These weaknesses can be fixed by either eliminating the escaping or dumping of the stack pointer when that is possible, or by using encryption, i.e., XOR-ing the dumped stack pointer with another secret we control. This is already done for setjmp in glibc.

3.3.3 Runtime Library

Most of the instrumentation by the above passes are added as intrinsic function calls, such as `cpi_ptr_store()` or `cpi_memcpy()`, which are implemented by Levee's runtime support library (a part of `compiler-rt`). This design cleanly separates the safe pointer store implementation from the instrumentation pass. In order to avoid the overhead associated with extra function calls, we ensure that some of the runtime support functions are always inlined. We compile these functions into LLVM bitcode and instruct `clang` to link this bitcode into every object file it compiles. Functions that are called rarely (e.g., `cpi_abort()`, called when a CPI violation is detected) are never inlined, in order to reduce the instruction cache footprint of the instrumentation.

3.3.4 Other Features and Limitations

Binary level pointers

Some code pointers in binaries are generated by the compiler and/or linker, and cannot be protected on the IR level. Such pointers include the ones in jump tables, exception handler tables, and the global offset table. Bounds checks for the jump tables and the exception handler tables are already generated by LLVM anyway, and the tables themselves are placed in read-only memory, hence cannot be overwritten. We rely on the standard loader's support for read-only global offset tables, using the existing `RTLD_NOW flag`.

Limitations

The CPI design described in Section 3.2 includes both spatial and temporal memory safety enforcement for sensitive pointers. However our current prototype implements spatial memory safety only. It can be easily extended to enforce temporal safety by directly applying the technique described in CETS [117] for sensitive pointers.

Levee currently supports Linux, FreeBSD and Mac OS user-space applications. We believe Levee can be ported to protect OS kernels as well. Related technical challenges include integration with the kernel memory management subsystem and handling of inline assembly.

CPI and CPS require instrumenting all code that manipulates sensitive pointers; non-instrumented code can cause unnecessary aborts. Non-instrumented code could come from external libraries compiled without Levee, inline assembly, or dynamically generated code. Levee can be configured to simultaneously store sensitive pointers in both the safe and the regular regions, in which case non-instrumented code works fine as long as it only reads sensitive pointers but doesn't write them.

Inline assembly and dynamically generated code can still update sensitive pointers if instrumented with appropriate calls to the Levee runtime, either manually by a programmer or directly by the code generator.

Dynamically generated code (e.g., for JIT compilation) poses an additional problem: running the generated code requires making writable pages executable, which violates our threat model (this is a common problem for most control-flow integrity mechanisms). One solution is to use hardware or software isolation mechanisms to isolate the code generator from the code it generates.

Sensitive data protection

Even though the main focus of CPI is control-flow hijack protection, the same technique can be applied to protect other types of sensitive data. Levee can treat programmer-annotated data types as sensitive and protect them just like code pointers. CPI could also selectively protect individual program variables (as opposed to types), however it would require replacing the type-based static analysis described in Section 3.2.1 with data-based points-to analysis such as DSA [97, 100].

Future MPX-based implementation

Intel announced a hardware extension, Intel MPX, to be used for hardware-enforced memory safety [82]. We believe MPX (or similar) hardware can be re-purposed to enforce CPI with lower performance overhead than our existing software-only implementation. This might reduce the higher overhead that we get when instrumenting some C++ applications with CPI. MPX provides special registers to store bounds along with instructions to check them, and a hardware-based implementation of a pointer metadata store (analogous to the safe pointer store in our design), organized as a two-level lookup table. Our implementation can be adapted to use these facilities once MPX-enabled hardware becomes available. We believe that a hardware-based CPI implementation can reduce the overhead of a software-only CPI in much the same way as HardBound [49] or Watchdog [115] reduced the overhead of SoftBound.

3.4 Evaluation

In this section we evaluate Levee's effectiveness, efficiency, and practicality. We experimentally show that both CPI and CPS are 100% effective on RIPE, the most recent attack benchmark we are aware of (Section 3.4.1). We evaluate the efficiency of CPI, CPS, and SafeStack on SPEC CPU2006, and find average overheads of 8.4%, 1.9%, and 0% respectively (Section 3.4.2). To demonstrate practicality, we recompile the FreeBSD base system plus over 100 packages with CPI/CPS/SafeStack and report results on several benchmarks (Section 3.4.3).

We ran our experiments on an Intel Xeon E5-2697 with 24 cores @ 2.7GHz in 64-bit mode with 512GB RAM. The SPEC benchmarks ran on an Ubuntu Precise Pangolin (12.04 LTS), and the FreeBSD benchmarks in a KVM-based VM on the same system.

	SafeStack	**CPS**	**CPI**
Average (C/C++)	0.0%	1.9%	8.4%
Median (C/C++)	0.0%	0.4%	0.4%
Maximum (C/C++)	4.1%	17.2%	44.2%
Average (C only)	-0.4%	1.2%	2.9%
Median (C only)	-0.3%	0.5%	0.7%
Maximum (C only)	4.1%	13.3%	16.3%

Table 3.1: Summary of SPEC CPU 2006 performance overheads

3.4.1 Effectiveness on the RIPE Benchmark

We described the security guarantees provided by CPI, CPS, and SafeStack based on their design in Section 3.2. To experimentally evaluate their effectiveness, we use the RIPE [173] benchmark. This is a program with many different security vulnerabilities and a set of 850 exploits that attempt to perform control-flow hijack attacks on the program using various techniques.

Levee deterministically prevents all attacks, both in CPS and CPI mode; when using only SafeStack, it prevents all stack-based attacks. On vanilla Ubuntu 6.06, which has no built-in defense mechanisms, 833–848 exploits succeed when Levee is not used (some succeed probabilistically, hence the range). On newer systems, fewer exploits succeed, due to built-in protection mechanisms, changes in the run-time layout, and compatibility issues with the RIPE benchmark. On vanilla Ubuntu 13.10, with all protections (DEP, ASLR, stack cookies) disabled, 197–205 exploits succeed. With all protections enabled, 43–49 succeed. With CPS or CPI, none do.

The RIPE benchmark only evaluates the effectiveness of preventing existing attacks; as we argued in Section 3.2, CPI renders all (known and unknown) memory corruption-based control-flow hijack attacks impossible.

3.4.2 Efficiency on SPEC CPU2006 Benchmarks

In this section we evaluate the runtime overhead of CPI, CPS, and SafeStack. We report numbers on all SPEC CPU2006 benchmarks written in C and C++ (our prototype does not handle Fortran). The results are summarized in Table 3.1 and presented in detail in Figure 3.3. We also compare Levee to two related approaches, SoftBound [116] and control-flow integrity [1, 187, 185].

CPI performs well for most C benchmarks, however it can incur higher overhead for programs written in C++. This overhead is caused by an abundance of pointers to C++ objects that contain virtual function tables—such pointers are sensitive for CPI, and so all operations on them are instrumented. The same is also true for `gcc`: it embeds function pointers in some of its data structures and then uses pointers to these structures frequently.

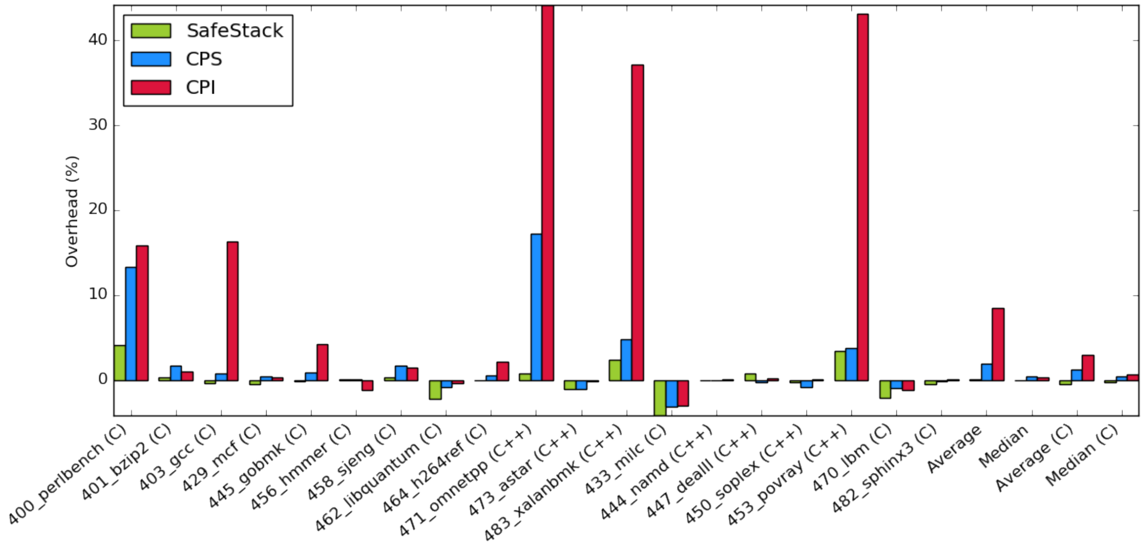


Figure 3.3: Levee performance for SPEC CPU2006, under three configurations: full CPI, CPS only, and SafeStack only.

The next-most important sources of overhead are `libc` memory manipulation functions, like `memset` and `memcpy`. When our static analysis cannot prove that a call to such a function uses as arguments only pointers to non-sensitive data, Levee replaces the call with one to a custom version of an equivalent function that checks the safe pointer store for each updated/copied word, which introduces overhead. We expect to remove some of this overhead using improved static analysis and heuristics.

CPS averages 1.2–1.8% overhead, and exceeds 5% on only two benchmarks, `omnetpp` and `perlbench`. The former is due to the large number of virtual function calls occurring at run time, while the latter is caused by a specific way in which `perl` implements its opcode dispatch: it internally represents a program as a sequence of function pointers to opcode handlers, and its main execution loop calls these function pointers one after the other. Most other interpreters use a `switch` for opcode dispatch.

SafeStack provided a surprise: in 9 cases (out of 19), it improves performance instead of hurting it; in one case (`namd`), the improvement is as high as 4.2%, more than the overhead incurred by CPI and CPS. Our hypothesis is that this is because objects that end up being moved to the regular (unsafe) stack are usually large arrays or variables that are used through multiple stack frames. Moving such objects away from the *safe stack* increases the locality of frequently accessed values on the stack, such as CPU register values temporarily stored on the stack, return addresses, and small local variables.

The SafeStack overhead exceeds 1% in only three cases, `perlbench`, `xalanbmk`, and `povray`. We studied the disassembly of the most frequently executed functions that use *unsafe stack* frames in these programs and found that some of the overhead is caused by inefficient handling of the *unsafe stack* pointer by LLVM's register allocator.

Benchmark	**FN**$_{UStack}$	**MO**$_{CPS}$	**MO**$_{CPI}$
400_perlbench	15.0%	1.0%	13.8%
401_bzip2	27.2%	1.3%	1.9%
403_gcc	19.9%	0.3%	6.0%
429_mcf	50.0%	0.5%	0.7%
433_milc	50.9%	0.1%	0.7%
444_namd	75.8%	0.6%	1.1%
445_gobmk	10.3%	0.1%	0.4%
447_dealII	12.3%	6.6%	13.3%
450_soplex	9.5%	4.0%	2.5%
453_povray	26.8%	0.8%	4.7%
456_hmmer	13.6%	0.2%	2.0%
458_sjeng	50.0%	0.1%	0.1%
462_libquantum	28.5%	0.4%	2.3%
464_h264ref	20.5%	1.5%	2.8%
470_lbm	16.6%	0.6%	1.5%
471_omnetpp	6.9%	10.5%	36.6%
473_astar	9.0%	0.1%	3.2%
482_sphinx3	19.7%	0.1%	4.6%
483_xalancbmk	17.5%	17.5%	27.1%

Table 3.2: Compilation statistics for Levee: FN$_{UStack}$ lists what fraction of functions need an unsafe stack frame; MO$_{CPS}$ and MO$_{CPI}$ show the fraction of memory operations instrumented for CPS and CPI, respectively.

Instead of keeping this pointer in a single register and using it as a base for all *unsafe stack* accesses, the program keeps moving the *unsafe stack* pointer between different registers and often spills it to the (safe) stack. We believe this can be resolved by making the register allocator algorithm aware of the *unsafe stack* pointer.

In Table 3.2 we show compilation statistics for Levee. The first column shows that only a small fraction of all functions require an *unsafe stack* frame, confirming our hypothesis from Section 3.2.3. The other two columns confirm the key premises behind our approach, namely that CPI requires much less instrumentation than full memory safety, and CPS needs much less instrumentation than CPI. The numbers also correlate with Figure 3.3.

Comparison to return address protections

Figure 3.4 compares the overhead of SafeStack with stack cookies and a shadow stack, showing average, minimum and maximum overhead numbers using the same SPEC 2006 benchmark. Stack cookies by default (`-fstack-protection`) has close to zero overhead on average and around 5% when the protection is applied to all functions (us-

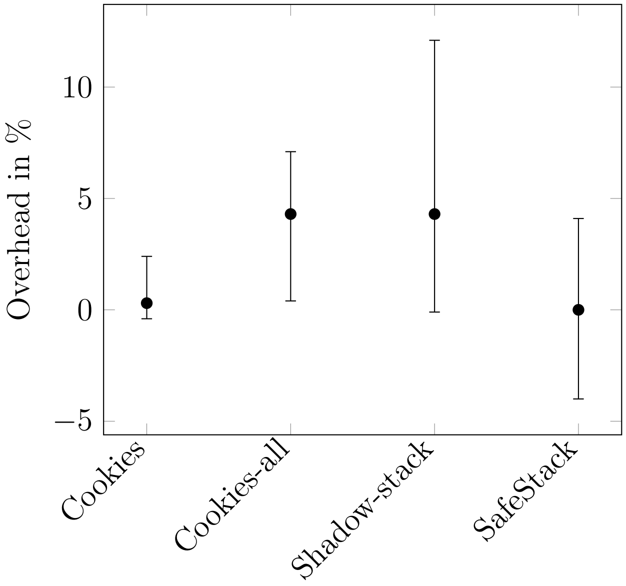


Figure 3.4: Comparison of the performance overhead of return address protections

ing the `-fstack-protection-all` flag). Our shadow stack implementation described in Section 2.6.1 has similar average overhead, but the worst case is higher. In comparison, the SafeStack transformation has zero overhead on average and while the maximum measured overhead was 4.1%, in many cases it improves the overhead with a similar amount.

Note that the SafeStack transformation provides a *strictly stronger* protection than both stack cookies (for all functions) or shadow stack. In particular SafeStack is more secure than stack cookies because it protects against direct overwrites and information leaks as well (by enforcing that no pointer is pointing to the safe stack outside of the safe stack). It is also more secure than shadow stacks, because while shadow stacks typically only protect saved return addresses, SafeStack protects all other saved or spilled registers (e.g., saved base pointers) and simple local variables as well. SafeStack can stop all return oriented programming exploits which rely on function or gadget chaining, while stack cookies and shadow stack do not.

SafeStack is now part of LLVM, its performance cost is negligible, it does not introduces compatibility issues, and provides stronger protection than cookies and shadow stacks. Therefore we believe it is a better alternative to stack cookies in being the default return address protection option in modern compilers.

Comparison to control-flow integrity protections

The average overhead for compiler-enforced CFI is 21% for a subset of the SPEC CPU2000 benchmarks [1] and 5-6% for MCFI [123] (without stack pointer integrity). CCFIR [185] reports an overhead of 3.6%, and BinCFI [187] reports 8.54% for SPEC CPU2006 to enforce a weak CFI policy with globally merged target sets. WIT [5],

Benchmark	SafeStack	CPS	CPI	SoftBound
401_bzip2	0.3%	1.2%	2.8%	90.2%
447_dealII	0.8%	-0.2%	3.7%	60.2%
458_sjeng	0.3%	1.8%	2.6%	79.0%
464_h264ref	0.9%	5.5%	5.8%	249.4%

Table 3.3: Overhead of Levee and SoftBound on SPEC programs that compile and run errors-free with SoftBound.

a source-based mechanism that enforces both CFI and write integrity protection, has 10% overhead.

At less than 2%, CPS has the lowest overhead among all existing CFI solutions, while providing stronger protection guarantees. Also, CPI's overhead is bested only by CCFIR. However, unlike any CFI mechanism, CPI guarantees the impossibility of any control-flow hijack attack based on memory corruptions. In contrast, there exist successful attacks against CFI [69, 46, 33]. While neither of these attacks are possible against CPI by construction, we found that, in practice, neither of them would work against CPS either. We further discuss conceptual differences between CFI and CPI in Section 3.5.

Comparison to memory safety protections

We compare with SoftBound [116] on the SPEC benchmarks. We cannot fairly reuse the numbers from [116], because they are based on an older version of SPEC. In Table 3.3 we report numbers for the four C/C++ SPEC benchmarks that can compile with the current version of SoftBound. This comparison confirms our hypothesis that CPI requires significantly lower overhead compared to full memory safety.

Theoretically, CPI suffers from the same compatibility issues (e.g., handling unsafe pointer casts) as pointer-based memory safety. In practice, such issues arise much less frequently for CPI, because CPI instruments significantly less pointers. Many of the SPEC benchmarks either don't compile or terminate with an error when instrumented by SoftBound, which illustrates the practical impact of this difference.

3.4.3 Case Study: A Safe FreeBSD Distribution

Having shown that Levee is both effective and efficient, we now evaluate the feasibility of using Levee to protect an entire operating system distribution, namely FreeBSD 10. We rebuilt the base system—base libraries, development tools, and services like `bind` and `openssh`—plus more than 100 packages (including `apache`, `postgresql`, `php`, `python`) in four configurations: CPI, CPS, Safe Stack, and vanilla. FreeBSD 10 uses LLVM/`clang` as its default compiler, while some core components of Linux (e.g., `glibc`) cannot be built with `clang` yet. We integrated the CPI runtime directly into

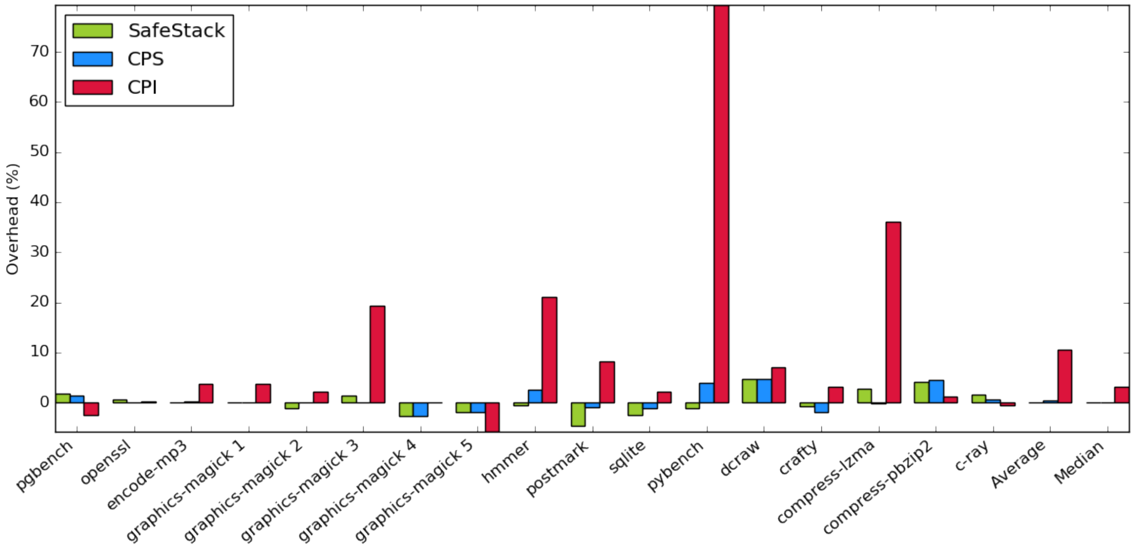


Figure 3.5: SafeStack/CPS/CPI performance overheads on FreeBSD (Phoronix)

the C library and the threading library. We have not yet ported the runtime to kernel space, so the OS kernel remained uninstrumented.

We evaluated the performance of the system using the Phoronix test suite [132], a widely used comprehensive benchmarking platform for operating systems. We chose the "server" setting and excluded benchmarks marked as unsupported or that do not compile or run on recent FreeBSD versions. All benchmarks that compiled and worked on vanilla FreeBSD also compiled and worked in the CPI, CPS and SafeStack versions.

Figure 3.5 shows the overhead of CPI, CPS and the SafeStack versions compared to the vanilla version. The results are consistent with the SPEC results presented in Section 3.4.2. The Phoronix benchmarks exercise large parts of the system and some of them are multi-threaded, which introduces significant variance in the results, especially when run on modern hardware. As Figure 3.5 shows, for many benchmarks the overheads of CPS and SafeStack are within the measurement error.

We also evaluated a realistic usage model of the FreeBSD system as a web server. We installed Mezzanine, a content management system based on Django, which uses Python, SQLite, Apache, and mod_wsgi. We used the Apache `ab` tool to benchmark the throughput of the web server. The results are summarized in Table 3.4.

The CPI overhead for a dynamic page generated by Python code is much larger then we expected, but consistent with suspiciously high overhead of the `pybench` benchmark in Figure 3.5. We think it might be caused by the use of some C constructs in the Python interpreter that are not yet handled well by our optimization heuristics, e.g., emulating C++ inheritance in C. We believe the performance might be improved in this case by extending the heuristics to recognize such C constructs.

Benchmark	SafeStack	CPS	CPI
Static page	1.7%	8.9%	16.9%
Wsgi test page	1.0%	4.0%	15.3%
Dynamic page	1.4%	15.9%	138.8%

Table 3.4: Throughput benchmark for web server stack (FreeBSD + Apache + SQLite + mod_wsgi + Python +Django).

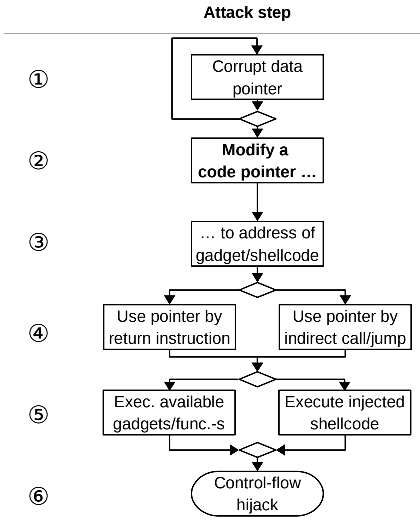
3.5 Comparison to Related Work

We relate CPI, CPS, and SafeStack to other hardening techniques. Figure 3.6 compares the design of the different protection approaches to our approach. As we are solely concerned with *control-flow hijack*s, the figure only contains the relevant attack steps from Figure 2.1. The table next to the figure lists the policies corresponding to each step and defense mechanisms enforcing these policies. For each one of them we point out their weaknesses along with their cost.

Step 1. Enforcing memory safety ensures that no dangling or out-of-bounds pointers can be read or written by the application, thus preventing the attack in its first step. Cyclone [83] and CCured [118] extend C with a safe type system to enforce memory safety features. These approaches face the problem that there is a large (unported) legacy code base. In contrast, CPI and CPS both work for unmodified C/C++ code. SoftBound [116] with its CETS [117] extension enforces *complete* memory safety at the cost of 2× − 4× slowdown. Tools with less overhead, like BBC [6], only protect the integrity of some object by enforcing some *approximation* of memory safety. LBC [73] and Address Sanitizer [146] detect continuous buffer overflows and (probabilistically) indexing errors, but can be bypassed by an attacker who avoids the red zones placed around objects. Write integrity testing (WIT) [5] provides spatial memory safety by restricting pointer writes according to points-to sets obtained by an over-approximate static analysis (and is therefore limited by the static analysis). Other techniques [51, 4] enforce type-safe memory reuse to mitigate attacks that exploit temporal errors (use after frees).

Step 2. CPI by design enforces spatial and temporal memory safety for a subset of data (code pointers) in Step 2 of Figure 3.6. Our Levee prototype currently enforces spatial memory safety and may be extended to enforce temporal memory safety as well (e.g., how CETS extends SoftBound). We believe CPI is the first to stop all control-flow hijack attacks at this step.

Step 3. Randomization techniques, like ASLR [130] and ASLP [89], mitigate attacks by restricting the attacker's knowledge of the memory layout of the application in Step 3. PointGuard [43] and DSR [18] (which is similar to probabilistic WIT) randomize


Attack step
①
Corrupt data pointer
②
Modify a code pointer ...
③
... to address of gadget/shellcode
④
Use pointer by return instruction
Use pointer by indirect call/jump
⑤
Exec. available gadgets/func.-s
Execute injected shellcode
⑥
Control-flow hijack

Policy	Mechanism	Stops all control-flow hijacks?	Avg. overhead
Memory Safety	SoftBound+CETS	**Yes**	116%
	BBC	No: sub-objects, reads not protected	110%
	LBC	No: protects red zones only	23%
	WIT	No: over-approximate valid sets	7%
Code-Pointer Integrity	CPI	**Yes**	8.4%
	CPS	No: valid code ptrs. interchangeable	1.9%
	SafeStack	No: precise return protection only	~0%
Randomization	ASLR, ASLP	No: vulnerable to information leaks	~10%
	PointGuard	No: vulnerable to information leaks	10%
	DSR	No: vulnerable to information leaks	20%
	NOP insertion	No: vulnerable to information leaks	2%
Control-Flow Integrity	Stack cookies	No: probabilistic return protection only	~2%
	CFI	No: over-approximate valid sets	20%
	WIT (CFI part)	No: over-approximate valid sets	7%
	DFI	No: over-approximate valid sets	104%
Non-Executable Data	HW (NX bit)	No: code reuse attacks	0%
	SW (Exec Shield, PaX)	No: code reuse attacks	few %
High-level policies	Sandboxing (SFI)	Isolation only	varies
	ACLs	Isolation only	varies
	Capabilities	Isolation only	varies

Figure 3.6: Summary of control-flow hijack defense mechanisms aligned with individual steps that are necessary for a successful attack. The figure on the left is a simplified version of the complete memory corruption diagram of Chapter 2.

the data representation by encrypting pointer values, but face compatibility problems. Software diversity [77] allows fine-grained, per-instance code randomization. Randomization techniques are defeated by information leaks through, e.g., memory corruption bugs [154] or side channel attacks [81].

Step 4. *Control-flow integrity* [1] ensures that the targets of all indirect control-flow transfers point to valid code locations in Step 4. All CFI solutions rely on statically pre-computed context-insensitive sets of valid control-flow target locations. Many practical CFI solutions simply include every function in a program in the set of valid targets [185, 187, 101, 162]. Even if precise static analysis was be feasible, CFI could not guarantee protection against all control-flow hijack attacks, but rather merely restrict the sets of potential hijack targets. Recent results [69, 46, 33] show that many existing CFI solutions can be bypassed in a principled way. CFI+SFI [184], Strato [183] and MIPS [122] enforce an even more relaxed, statically defined CFI policy in order to enforce software-based fault isolation (SFI). CCFI [107] encrypts code pointers in memory and provides security guarantees close to CPS. Data-flow based techniques like data-flow integrity (DFI) [34] or dynamic taint analysis (DTA) [141] can enforce that the used code pointer was not set by an unrelated instruction or to untrusted data, respectively. These techniques may miss some attacks or cause false positives, and have higher performance costs than CPI and CPS. Stack cookies, CFI, DFI, and DTA protect control-transfer instructions by detecting illegal modification of the code pointer whenever it is used, while CPI protects the load and store of a code pointer, thus preventing the corruption in the first place. CPI provides precise and provable security guarantees.

Step 5. In Step 5, the execution of injected code is prevented by enforcing the non-executable (NX) data policy, but code-reuse attacks remain possible.

Step 6. High level policies, e.g., restricting the allowed system calls of an application, limit the power of the attacker even in the presence of a successful control-flow hijack attack in Step 6. Software fault isolation (SFI) techniques [108, 59, 35, 177, 184] restrict indirect control-flow transfers and memory accesses to part of the address space, enforcing a sandbox that contains the attack. SFI prevents an attack from escaping the sandbox and allows the enforcement of a high-level policy, while CPI enforces the control-flow inside the application.

Part II

Automated Test Generation for Finding Memory Corruption Bugs

Chapter 4

A Survey of Automated Security Testing Research

This chapter presents an overview of existing automated test generation approaches for finding memory corruption bugs and analyzes their strengths and weaknesses. We show that all automated testing techniques can be considered as search over the input space of a target program to identify inputs that trigger bugs. More directly, this means that we search for inputs that trigger new behavior (new states or new coverage) in the program. In other words, we sample the input space in order to explore the execution state space, and find as many distinct points in it as possible. The more new states we can reach, the more erroneous states we can find. In the following text we will often refer to inputs that trigger new coverage (e.g., new paths or new code lines) simply as *interesting* inputs.

We will discuss *random* and *systematic* approaches and also their combinations. Random techniques, like black-box fuzzing, explore the input space randomly, however the search is typically guided by some feedback information (gray-box fuzzing). Due to randomness, we cannot make any guarantees about these techniques. Systematic (white-box) approaches, like symbolic execution, are more complex and costlier than random ones, but usually also more powerful. They explore the execution space by systematically targeting specific points (e.g., a particular path) they want to reach with the generated input. This, in theory, allows them to offer deterministic guarantees, such as being able to discover all possible paths in the program, given enough time. In practice, however, due to the size of the execution state space, it is hard to make strong guarantees about either class of approaches.

4.1 Fuzzing

4.1.1 Black-box Fuzzing

The idea of random black-box testing is to simply sample the input space without any guidance, in other words running the target with random inputs. As a generic testing approach it was under active study in the 1980's, but other than the observation that it may often be more cost effective than systematic techniques, no conclusive results were established at that time [57]. Using the idea in the security (and reliability) context started with an experiment by Miller et al. [112], in which they fed random character strings to Unix applications and found that they can be easily crashed that way. This is where the term "fuzzing" originates from.

Following the Miller experiment, the approach quickly gained traction and became the most effective practical method of finding security bugs automatically [125], [157], [158], and it has been a huge success ever since. Fuzzing became standard practice in the software and security industry. Today it is a mandatory element of the software development life-cycle at all major software companies, such as Microsoft [161], Google [160] or Adobe [159]. Random fuzzing has been widely used in various research works to find security bugs. Custom built fuzzers have been used against a diverse set of targets, from virtual machines [106], through security protocols [165] to certificates [28]. In recent years fuzzing as a generic approach for finding bugs has also been studied more formally [174], [135] [37], [189], [155], [151], [26].

Domain-specific fuzzers target a specific input format, which means they require an (under-specified) grammar of the input language [86]. Using the grammar to parse the inputs, random mutations can be carried out on an abstract syntax tree level. Alternatively, random (semi-valid) sentences can be generated from the grammar [176, 76]. However, even a completely generic fuzzer can be effective in finding bugs automatically, saving the manual effort of grammar specification. This is done by taking existing valid inputs and making small mutations on them, for instance, by randomly flipping some bits. Mutating a good input sample is of course more effective than feeding completely random character strings to programs. Algorithm 4.1 shows the pseudo-code for black-box fuzzing.

Algorithm 4.1: Black-box Fuzzing

Input: Initial test suite *TestSuite*.

Output: Set of crashing inputs *Crashers*.

```
while $TestSuite \neq \varnothing$ do
    $t \leftarrow \textsc{SelectFrom}(TestSuite)$        ▷ Take a test vector out of the test suite.
    for each $t' \in \textsc{Mutations}(t)$ do
        $\textsc{Run}(t')$
        if $\textsc{Crashed}(t')$ then
            $Crashers \leftarrow Crashers \cup \{t'\}$
```

Note that when we are looking for memory corruption bugs, we do not have the *oracle problem* of functionality testing, which is the problem of automatically deciding whether a given behavior of the program is correct or not. Triggering a memory corruption bug, even without precise detection mechanisms, typically causes the program to crash, which can be used as an oracle to decide whether a test passed or failed.

Black-box fuzzing is fast but has little coverage increasing power. It has practically no overhead as the target can be run natively, but it can only work well if an initial test suite with good coverage is already available. It has the potential to trigger bugs along the paths exercised by the initial test vectors, but it cannot increase coverage significantly beyond the coverage of the initial test set.

4.1.2 Coverage-guided Fuzzing

The primary limitation of black-box fuzzing is that it explores the input space only "around" the inputs of the initial test suite. Let us call two inputs, or two points in the search space, *neighbors*, if we can get from one to the other through one mutation step (e.g., a bit flip, or byte change, etc.). Black-box fuzzing only explores immediate neighbors of the starting inputs, because the blind exploration of multi-step neighbors is impractical. To make it practical, we need a criterion for selecting neighbors that are interesting enough to continue searching in their neighborhoods. Coverage-guided fuzzing determines a neighbor to be interesting if it covers new states (e.g., a new branch in the code), that was not covered before. This idea is captured in Algorithm 4.2. It only differs from black-box fuzzing in the last two lines, which adds the interesting mutants that cover new states back to the test suite for further mutation. By adding the inputs covering new points in the state space, we create new base points for further exploration. This allows us to incrementally discover more and more of the search space, as long as new interesting inputs can be reached through neighboring mutations. Fuzzers that use some feedback information to guide their search, such as coverage feedback, are often called gray-box fuzzers.

Algorithm 4.2: Coverage-guided Fuzzing

```
Input: Initial test suite $TestSuite$.
Output: Set of crashing inputs $Crashers$.
while $TestSuite \neq \varnothing$ do
    $t \leftarrow$ SelectFrom($TestSuite$)
    for each $t' \in$ Mutations($t$) do
        Run($t'$)
        if Crashed($t'$) then
            $Crashers \leftarrow Crashers \cup \{t'\}$
        if NewCoverage($t'$) then
            $TestSuite \leftarrow TestSuite \cup \{t'\}$
```

AFL

AFL (American Fuzzy Lop) [180] was the first fuzzer to demonstrate how effective coverage-guided fuzzing can be. For instance, AFL was able to generate valid JPEG files without an initial test suite, by fuzzing a JPEG parsing library for a few days [182]. It is arguably the most popular and most widely used fuzzer today. Since its release, a continuously growing number of security critical bugs have been found with it in real world software [180]. Due to its success, other tools have borrowed its approach as well, such as LibFuzzer [147] or Syzkaller[168].

AFL uses (counted) edge coverage as a coverage metric. An execution is considered new coverage if it traverses an edge in the control-flow graph that has not been traversed before, or it traverses an edge more times then before. Instead of counting any increment in edge traversals, it is more common to create buckets of counts and consider an input *interesting* if it lands in a new bucket. Buckets typically increase exponentially after a threshold, e.g., $\{1, 2, 3, 4-7, 8-15, 16-31, 32-63, 64-\infty\}$.

There are two primary reasons why such a relatively simple random approach can work so well. One is that using coverage is a good way to distinguish meaningful inputs from redundant ones. The other one is that one iteration (one mutation and execution step) is very fast. Technically the only overhead compared to native execution is the overhead of the instrumentation that keeps track of coverage, which is typically around 50%. This is multiple orders of magnitude smaller than the overhead of some of the more systematic approaches, like symbolic execution, which can be $1000\times$ [68, 66]. So while black- or gray-box fuzzing is not as powerful as systematic approaches, in some cases they can compete in effectiveness due to their speed.

AFLFast [26] is a recent extension of AFL. When AFL takes a seed input to fuzz from the test suite, it usually creates a relatively high number (~80k) of mutations from it. The main contribution of AFLFast is the introduction of a heuristic that tells how much an input selected at step 3 of Algorithm 4.2 should be fuzzed (how many mutations should be created from it). Without this heuristic, after fuzzing a number of inputs, many of the mutations will exercise some *high-frequency paths*, such as the one taken when an invalid file is rejected early on. The intuition behind their heuristic is that these high-frequency files should not be fuzzed as much as the low-frequency ones. Low-frequency ones usually represent the more interesting paths and the likelyhood of generating a low-frequency mutation form another low-frequency seed is higher than from a high-frequency seed. So AFLFast starts fuzzing inputs for only a short time in the beginning, and then increase the number of mutations only for the low-frequency inputs.

The weakness of coverage-guided fuzzing is that, while it is capable of increasing the coverage, it does it relatively ineffectively. It is able to discover interesting points in the input space multiple mutation steps away, but the mutations themselves are carried out completely blindly. In other words, it is able to detect when a random mutation exercises a new path, but it has no information about what part of the input to mutate, or how to mutate it in order to trigger a new path.

4.1.3 Taint-guided Fuzzing

The blind fuzzers discussed so far mutate all parts of an input with the same probability. Taint-guided fuzzers use taint (data-flow) tracking to decide which parts of a given input to mutate more than others. Dynamic taint analysis can be used to figure out which bytes of the input influence some of the more critical parts of the code. This idea was used in Flayer [56], Smart Fuzzer [95], BuzzFuzz [64], TaintScope [170], TaintFuzz [102] and others [16].

In particular, BuzzFuzz uses taint-tracking to identify what parts of the input flow into potentially dangerous library function calls (e.g., `malloc` or `strcpy`). The list of these functions, however, are provided by the user. The input bytes that flow into the arguments of the selected functions will be fuzzed more. TaintScope uses the same approach, but in addition, it also automatically identifies checksum checks, based on the taint information. When it detects that the code computes a checksum on a large part of the input and compares it to a checksum field, it automatically rewrites that check in the code to force it to always pass. This is similar to what Flayer does, which is a tool that can help the user (security tester) identify such checks and "patch them out" manually. Removing these checks makes the target more fuzzable. Dowser [72] is another related tool, as it uses taint tracking to identify which bytes flow into code regions where it is more likely that a buffer overflows can happen. Since it uses symbolic execution instead of fuzzing, we cover it in the next section.

In summary, these tools can be used to significantly reduce the search space, and thus the number of mutations necessary to trigger a bug. But primarily, they only help in triggering errors along the paths exercised by the initial inputs. They do not increase the effectiveness of the mutations in finding completely new paths. Further, while the taint feedback allows us to narrow down *what* part of the input to fuzz, it does no provide information about *how* to mutate the selected bytes.

4.2 Dynamic Symbolic Execution (DSE)

Dynamic symbolic execution (DSE) or *concolic execution* [32] is a systematic approach. Similar to fuzzing, it generates new interesting inputs from existing ones, but in a systematic way, by looking at what happens precisely during the execution. This is why it is also called *white-box fuzzing* [68]. (In this text we will use the term fuzzing only to refer to black- and gray-box fuzzing.) New inputs are computed in two steps. First it runs the program with the initial input and collects the constraints on the input that define the triggered execution path. Then it negates one of those constraints to represent an alternate path and uses a constraint solver to find a satisfying input for it. So the first part creates the specific targets (i.e., paths) for the test generation and the actual input search is carried out by the constraint solver.

Algorithm 4.3: Dynamic symbolic execution, based on SAGE's generational search.

```
Input: Initial test suite $TestSuite$.
Output: Updated test suite $TestSuite$, and set of crashing inputs $Crashers$.
function DSE($TestSuite$)
    while $TestSuite \neq \varnothing$ do
        $t \leftarrow$ SelectFrom($TestSuite$)
        for each $t' \in$ NewInputs($t$) do
            Run($t'$)
            if Crashed($t'$) then
                $Crashers \leftarrow Crashers \cup \{t'\}$
            $TestSuite \leftarrow TestSuite \cup \{t'\}$
function NewInputs($t$)
    $Children \leftarrow \varnothing$
    $pc \leftarrow$ ExtractPathCondition($t$)                ▷ Symbolic execution.
    for $i \leftarrow min\_level(t)$ to $n$ do           ▷ Skip already negated branches.
        $pc' \leftarrow (c_0 \wedge c_1 \cdots \wedge \neg c_i)$
        $t_{child} \leftarrow$ SMTSolve($pc'$)       ▷ Returns UNSAT or an input assignment.
        if $t_{child} \neq$ UNSAT then
            $min\_level(t_{child}) \leftarrow i$                ▷ Set negated branch level.
            $Children \leftarrow Children \cup \{t_{child}\}$
    return $Children$
```

Algorithm 4.3 shows the pseudo-code, modeled after SAGE [68]. The main loop in the DSE() function is similar to Algorithm 4.2, but the random Mutations() function is replaced with the systematic NewInputs(), defined below the DSE() function.

The ExtractPathCondition() function carries out the symbolic execution to extract the formula containing the input constraints, called the *path condition*. This is done by maintaining an additional symbolic state during the execution of the program. The name *concolic*, comes from concrete and symbolic execution. In the symbolic state each variable is an expression, encoding the computation made on the input variables. For instance, if we have an input variable `a`, we run the program with the concrete value `a=42` and the symbolic value `a=`α. If we assign `b=a+3`, the concrete value of `b` will be `45`, while the symbolic value will be $\alpha + 3$. Whenever the execution reaches a conditional branch, the path condition is updated with the constraining predicate of the taken branch. For instance if the execution passes an `if (b == 50)` branch, the path condition will be $c_0 \wedge \cdots \wedge c_{n-1} \wedge (\alpha + 3 = 50)$. The ExtractPathCondition() returns the extracted formula in the conjunctive normal form: $pc = (c_0 \wedge c_1 \cdots \wedge c_n)$.

Next, in order to generate inputs which will make the program take alternative paths, we take the prefixes of the path condition formula and negate the last clauses. We start from the prefix that we have not explored before, to avoid redundant searches. The $min\_level(t)$ defines this smallest prefix for each input. A constraint solver, given this new expression, tries to find a solution that satisfies it, which can be used as the new input. If the constraint solver returns a solution for the modified path condition, we save and return it.

DSE tools

DSE received a great deal of interest during the last decade or so [32]. Because of the large number of different symbolic execution tools and papers on the topic, we do not intend to cover every existing tool in this chapter, as other detailed surveys exist on this topic [14]. We only mention a selected few systems.

Originally, symbolic execution is a static analysis technique. DART [67] introduced the idea of dynamic symbolic execution, which means executing only a single path symbolically, driven by a concrete execution, as described above. CUTE [145] extended DART's approach in multiple aspects, most importantly it supported treating pointers symbolically, which DART did not. Being able to set constraints for pointers, allowed the generation of data structures with pointers in them as well. The DART and CUTE algorithm used a DFS strategy for selecting which branch to negate next, which is different from the generation search strategy of SAGE [68] that we show in Algorithm 4.3. While DFS targets a single path after one symbolic execution, SAGE's strategy maximizes the number of new test cases generated from one (costly) symbolic execution, by targeting all diverging paths. Another difference is that while DART and CUTE works on C code, SAGE works on x86 binaries. SAGE was developed and used internally at Microsoft and reportedly found several critical bugs in Microsoft products.

KLEE [31] is an open-source symbolic execution tool built on top of LLVM [98], which uses a slightly different design than the tools discussed so far. While the above tools always reason about a single execution path at a time, KLEE attempts to execute multiple paths simultaneously in a single run. Whenever the execution reaches an input dependent branch, KLEE clones and forks the execution state. While this means that a large number of different states need to be kept track of, a significant amount of the state can be shared without duplicating it in the memory. Using this approach, the repeated re-execution of the same path prefixes can be avoided. KLEE models the memory with bit-level accuracy, employs search heuristics to get high code coverage, and also uses a variety of constraint solving optimizations.

Mayhem [36], a tool developed at CMU, also works on binaries. It uses both SAGE's single path and KLEE's multiple path strategy to be able to balance its speed and memory requirement more flexibly. It starts with the multiple path strategy, but when a threshold is reached in memory usage, a checkpoint is made and single path explorations continue from the saved states. Mayhem recently won the DARPA Cyber Grand Challange (CGC) competition.

Symbolic execution, other than test case generation for bug finding, can also be used for automatically generating exploits [12] or generating input filters to block a given exploit [42, 30]

Taint-guided dynamic symbolic execution

Data-flow tracking (or dynamic taint analysis) is often used together with DSE as well. Taint analysis can be useful to decide which parts of the program to run symbolically, or which part of the input to make symbolic for DSE. Lanzi et al. [95] proposed a fuzzer that used taint tracking on binaries in order to build path condition formulas for a preliminary implementation of symbolic execution. Mayhem [36], the DSE system for binaries, also runs the target only with taint tracking, and sends to the symbolic executer only the basic blocks that contain tainted instructions.

Dowser [72] uses taint tracking to identify which bytes flow into code regions where it is more likely that a buffer overflows can happen. Specifically, they first statically analyze the code to find loops with unchecked loop indices and array dereferences. Then the taint analysis identifies which input bytes influence these array indices, so that the following symbolic execution only treats those bytes symbolically. DIODE [152] is a tool to find a typical bug type, namely when `malloc` gets called with the wrong allocation size due to an integer overflow. It also uses taint tracking as its first stage to find memory allocation sites where the input influences the size of the allocation. Then it uses DSE by treating those bytes symbolically, in order to try to generate an input that triggers the bug.

Strengths and weaknesses of DSE

DSE is powerful due to its systematic search, but this power comes with costs and trade-offs. In general, the problem with even the latest state-of-the-art DSE systems is that they do not scale to larger programs and are often impractical in the real-world. In their recent analysis, the creators of Angr [151] compared fuzzing and DSE, using the most advanced state-of-the-art techniques in both cases. In their experiment, using the CGC (Cyber Grand Challange) benchmark, the fuzzer AFL found 68 vulnerabilities, while DSE only found 23. To quote the authors: "*The difference between the results of the various dynamic symbolic execution approaches are surprising. One might reasonably expect DSE to identify roughly as many vulnerabilities as symbolically-assisted fuzzing, and more than fuzzing alone. In reality, fuzzing identified almost three times as many vulnerabilities. In a sense, this mirrors the recent trends in the security industry: symbolic analysis engines are criticized as impractical while fuzzers receive an increasing amount of attention. However, this situation seems at odds with the research directions of recent years, which seem to favor symbolic execution*" In the following points we hypothesize some of the main reasons why DSE might perform relatively worse than less systematic approaches.

The most evident weakness of DSE is that it is slow. One reason is that maintaining the symbolic state has high overhead. The overhead depends on the implementation, but e.g., SAGE slows down the execution by 1000× [68, 66]. This is only the overhead of running the program symbolically in order to extract the path constraints. These are handed over to a separate constraint solver that computes the actual input satisfying the targeted path condition. Constraint solving can also be a bottleneck [32]. Often times, queries generated by some programs will not only make the solver dominate the runtime, but because they cannot be solved in a reasonable time, the solver times out without a solution. Longer paths create larger path condition formulas and typically the time to solve them grows exponentially.

Due to this, DSE often only finds shallow bugs. This observation was first made in the SAGE paper [68]. They found that all crashing inputs were only a few generations away from the seed file. The Angr study [151] also showed that the bugs found by DSE represented only relatively short paths, while the ones found by fuzzing represented various lengths. Further, the analysis points out that “symbolic execution (including Veritesting) covered an average of 330 blocks per binary (with a median of 260), while fuzzing covered 689 (with a median of 402).”

To mitigate the problem of complex formulas, DSE systems try to exploit that there is typically a lot of redundancy between the path conditions given to the solver, so they usually apply techniques to make solving more incremental. The most typical approach, used by CUTE [145], KLEE [31], Mayhem [36] and others, is to cache solver queries. The path conditions are split to independent formulas (that do not share variables) and the results of the independent queries are memoized. If a formula that has been queried before appears again in a path condition, the cache can be used to quickly determine that it is unsatisfiable, or we can also try to substitute the cached solution in the path condition.

Caching can greatly improve the performance of constraint solving. The above mentioned approach, however, only works for independent constraints. The cached solution $[x = 1, y = 2]$ for $(x < y)$ cannot be used to help solving $(x < y) \wedge (42 < x)$. Modern constraint solvers such as Z3 [48] support incremental solving for similar constraints, even if they are dependent, by keeping track of *lemmas*. The cost of using this feature is that the solver needs to keep its state between queries, it requires significantly more memory, and it is harder to parallelize. To the best of our knowledge, no current DSE system uses the incremental feature of SMT solvers.

The main strength of DSE is that the generated inputs are likely to trigger new coverage. DSE trades off speed for this property: while one iteration takes a long time, the probability that the generated input will reach a new state is much higher than with the blind exploration of the input space. However, even if the solver finds a satisfying input for the targeted path condition, it is not guaranteed that the input will take the targeted path. The constraints generated by symbolic execution often do not capture the path condition perfectly due to imperfections of modeling. This means that the generated path constraint will not describe the targeted path precisely and the generated input will diverge. We can talk about two kinds of modeling, where both can be source of imprecision: memory modeling and the environment modeling. The former is how we model the symbolic variables, which also has an impact on the overhead. For instance, using a fixed-width bit-precise integer model and bitvector theory in the SMT solver, instead of mathematical integers, has additional cost. The same goes for modeling floating point numbers or pointers and using theory of arrays in the solver. Consequently, in practice, due to scalability issues and due to the limitations of SMT/SAT solvers, some variables are modeled with approximations or not modeled at all. Cedar et al. [32] suggests that "The trade-off between precision and scalability should be determined in light of the code being analyzed [. . .] and the exact performance difference between different constraint solving theories."

The other potential source of imprecision is the modeling of the environment, which is due to the parts of the program that cannot be executed symbolically, such as external library calls or system calls. In order to propagate symbolic information through these external parts of the execution, the user needs to provide models. Due to the need of manual effort, DSE systems do not provide such models for everything, which causes imprecision. This means that path divergence due to imprecision is unavoidable in practice. For instance the SAGE paper [68] reported a divergence rate of 60%. This means that in 40% of the cases the input returned by the SMT solver exercised the path predicted by the solved path condition. Note, that relative to fuzzing, this is a good ratio.

In summary, DSE is on the opposite extreme of the speed vs. precision trade-off, than fuzzing. However, in different scenarios, different trade-offs might work better. Neither approach is clearly superior than the other.

4.2.1 Constraint Solvers

The path conditions extracted by DSE are formulas of *Satisfiability Modulo Theories* (SMT) and solved by SMT solvers. SMT represents a decision problem for logical formulas with respect to some background theory, such as the theory of bit-vectors or integers. Modern DSE systems (e.g., KLEE), typically use bit-vectors instead of integers in order to accurately reason about bitwise operations, casts, and arithmetic overflows. As discussed in the previous section, using bit-level accurate constraints means trading off some performance for precision to avoid path divergence. The classical approach solvers take for solving bit-vector formulas is that they translate the higher level SMT formula into a boolean SAT formula and solve it with a SAT solver. This is called *bit-blasting*.

SAT and SMT solving techniques can also be categorized into systematic and random approaches. Most of the state-of-the-art solvers, such as CVC4 [15] and Z3 [48] use systematic algorithms, based on backtracking. They use the Conflict-Driven Clause Learning (CDCL) [20] algorithm, which extends the Davis-Putnam-Logemann-Loveland (DPLL) [65] algorithm. The systematic backtracking based algorithms are sound and complete. Soundness means that if an answer is returned (i.e., a satisfiable assignment or unsatisfiable), it is correct. Completeness means that it always returns an answer (given enough time).

On the other hand there exist randomized, heuristic algorithms, which use stochastic local search to find a solution. These local search algorithms try to iteratively get closer to the solution by starting at some location of the given search space and subsequently move to a neighboring location, where each location has only a relatively small number of neighbors, and each move is determined by a decision based on local knowledge only. These algorithms are sound but incomplete, so it is not guaranteed to return an answer. This means that even if a satisfying assignment exists, it is not guaranteed to be found, but in practice even a systematic algorithm is not guaranteed to find a solution given a time limit. Incomplete algorithms also cannot prove unsatisfiability, but for test generation purposes, we are mostly interested whether we can find a satisfying solution within a given time limit.

We cover these incomplete algorithms in depth in the following subsection, since they will be relevant for the next chapter of the dissertation.

Local search based solvers

This section discusses *stochastic local search* (SLS) based solvers. We describe two SLS based SAT solvers and a recent SMT solver.

All local search based techniques can be described by the following three elements:

- A *scoring function* assigns a score to a point in the search space, representing how good that point is or how close it is to a solution.
- A *neighbor function* defines for each point what are the neighboring points where moves can be made.
- A *search strategy* specifies how we move between neighboring points, based on the scoring function.

SLS based SAT solving. GSAT [144] was the first SLS based SAT solver. We define the GSAT algorithm, by defining the three elements of any local search based system. Most solvers, including GSAT expect SAT formulas to be represented in conjunctive normal form (CNF), $F = c_1 \wedge c_2 \wedge \cdots \wedge c_n$, where each clause is the disjunction of positive or negative Boolean literals. The search space is the space of possible truth assignments to the boolean variables in the formula. GSAT scores a truth assignment by the number of unsatisfied conjunctive clauses. The lower the score, the better. The neighbors of a truth assignment are the ones in which one of the variable's assignment is flipped. Finally the used search algorithm is the most basic one, characterized variously as greedy, best-first search, or steepest ascent hill climbing. As the pseudo-code shows in Algorithm 4.4, we simply move towards the best possible neighbor. Random restarts are made when the search gets stuck in local minima.

WalkSAT [143], modifies GSAT in order to be more robust against local minima and plateaus, where all neighbors have the same score. It does this by introducing an extra step of random walk. With a fixed probability, instead of taking the move with the best score it takes a random move. This is shown in Algorithm 4.5.

The modified algorithm achieves substantially better performance than GSAT, because the tendency of stagnation is reduced by the random walks. Another modification of WalkSAT that improved the search even further is to first randomly select a clause C

Algorithm 4.4: GSAT algorithm

```
Input: CNF formula $f$, $max\_restarts$, $max\_steps$
Output: Satisfying assignment $s$ or UNSAT
for $i \leftarrow 1$ to $max\_restarts$ do
    $s \leftarrow$ RandomAssignment($f$)
    for $j \leftarrow 1$ to $max\_steps$ do
        if $f|_s =$ SAT then
            return $s$
        $s \leftarrow$ GetOneWithBestScore(Neighbors($s$))
return UNSAT
```

Algorithm 4.5: WalkSAT algorithm

```
Input: CNF formula $f$, $max\_restarts$, $max\_steps$, probability of doing random walk $p$
Output: Satisfying assignment $s$ or UNSAT
for $i \leftarrow 1$ to $max\_restarts$ do
    $s \leftarrow$ RandomAssignment($f$)
    for $j \leftarrow 1$ to $max\_steps$ do
        if $f|_s =$ SAT then
            return $s$
        if Random($[0, 1)$) $< p$ then
            $s \leftarrow$ PickRandomlyFrom(Neighbors($s$))
        else
            $s \leftarrow$ GetOneWithBestScore(Neighbors($s$))
return UNSAT
```

that falsifies F and only consider the variables that appear in C. This step dynamically narrows down the set of possible neighbors to the ones in which only the variables of C are flipped.

SLS based SMT solving. Fröhlich et al. [63] recently published a local search based solver that works on the SMT bit-vector theory level. Working on the theory level instead of reducing the problem to SAT has the advantage that the structural information of the high level formula is not lost.

We define the SLS-SMT algorithm by defining the scoring function, neighbor function, and the search strategy. For the sake of brevity, let us consider bit-vector formulas with conjunctions only, as typically path conditions extracted by DSE, like SAGE, do not contain disjunctions. An SMT formula $F = a_1 \wedge a_2 \wedge \cdots \wedge a_n$ is a conjunction of (potentially negated) assertions, where assertions are either boolean variables or relations $(=, \leq)$ between two bit-vector expressions. For example, $F = (x + 3 \leq y - 5) \wedge \neg(x = 42)$, where x and y are bit vectors of size 32.

We define the score s, of a variable assignment α for a formula $F = a_1 \wedge \cdots \wedge a_n$ recursively as follows. The score always yields a value over $[0, 1]$.

$$s(F, \alpha) = \frac{1}{n} \cdot (s(a_1, \alpha) + \cdots + s(a_n, \alpha))$$

If the assertion is a boolean variable, then the score is 0 if it evaluates to false, and 1 if it evaluates to true. If the assertion is an equality relation between two expressions t_1 and t_2, with variable assignment α, then

$$s(t_1 = t_2, \alpha) = 1 - \frac{h(t_1|_\alpha, t_2|_\alpha)}{n}$$

where h is is the Hamming distance and n is the size of the bit vectors. If the assertion is a less or equal relation, then

$$s(t_1 \leq t_2, \alpha) = 1 - \frac{abs(t_2|_\alpha - t_1|_\alpha)}{2^n}$$

The score of a variable assignment evaluates to 1 iff α is a satisfying assignment for F.

We define the neighbors of a variable assignment similarly to the SAT solvers, by flipping boolean variables or individual bits in bit-vector variables. Finally, the high level search algorithm used by SLS-SMT is very similar to the one used by WalkSAT, with some additional tweaks and heuristics.

Local search based constraint solvers have not received much attention since WalkSAT, as research focused on systematic algorithms. SLS-SMT however demonstrated that SLS based SMT solving can compete with backtracking based bit-vector solvers, even though there is still a gap between their performance in general. In some practical problem instances, however, SLS-SMT outperformed the state-of-the-art Z3 solver, particularly on benchmarks containing path conditions extracted by symbolic execution from real-world software. This suggests that there are problem instances, for which randomized algorithms can be more cost effective than systematic ones.

4.3 Search-Based Software Testing (SBST)

The idea of *search-based software testing* (SBST) is to use local search algorithms for generating test data [110, 7, 109]. SBST has primarily been used for unit test generation. Miller et al. [113] were the first to use local search for generating floating-point test data that exercises a particular path in the program. Subsequent work on SBST [90, 91, 61] focused on the problem of finding test data for a particular program element, such as a statement or branch, instead of a path. They target and try to generate inputs to cover one node in the control-flow graph at a time (independently of each other).

Predicate	Branch distance B
$x = y$	if $abs(x-y) = 0$ then 0 else $abs(x-y) + K_=$
$x \neq y$	if $abs(x-y) \neq 0$ then 0 else $K_{\neq}$
$x < y$	if $x - y < 0$ then 0 else $(x-y) + K_<$
$x \leq y$	if $x - y \leq 0$ then 0 else $(x-y) + K_{\leq}$

Table 4.1: Branch distance according to Wegener et al. [172]

Scoring function. The most often used *score function* used by SBST tools to reach and cover a branch was introduced by Wegener et al. [172]. It evaluates an input against the target branch using two measures, the *approach level* and the *branch distance*. The approach level represents the "node level" distance; the number of the target's control dependent branch nodes that were not covered by the input's path, i.e., the number of additional nested branches that need to be passed to reach the target.

On the node where the execution diverges from the path to the target branch, the branch distance tells how "far" the test case is to turn the divergent branch over to the target path. Table 4.1 defines its value for different relational predicate types.

The constant K is added when the undesired branch is taken. The final score function for a test input t and target branch b is the approach level A and the branch distance B normalized to $[0, 1]$. Score 0 means we reached the target branch.

$$Score(t, b) = A(t, b) + norm(B(t, b)) \tag{4.1}$$

Search strategies. Different tools use different search algorithms, such as hill climbing, alternating variable method (AVM) [90], genetic algorithm [137], or simulated annealing [163].

Simulated annealing makes hill climbing more stochastic. It takes random moves towards points with better scores, but to avoid getting stuck in local minima, it also allows moves that worsen the score with some probability. This probability depends on the score difference: the worse the new score would be, the less likely the move is taken (this idea comes from the Metropolis algorithm). Further, the probability is higher in the beginning of the search and is lowered with time, to allow the search to converge to the (hopefully) global minimum. We decrease the probability of taking "downhill" moves according to a *temperature schedule*, as shown in Algorithm 4.6.

The *alternating variable method* (AVM) does the hill climbing in two phases. Suppose the input consists of multiple variables (e.g., different arguments of the function under test). AVM first decides which variable to modify by trying some exploratory moves, such as adding ±1 to integer variables. If any of these moves improves the score, it takes the move and starts a pattern search, using that variable. During the pattern search it applies increasingly larger changes to the choosen variable as long as the score is improving. Once it stops improving, the process starts over. The pseudo code is shown in Algorithm 4.7.

Algorithm 4.6: Simulated Annealing

```
Input: Initial point $s$ in the search space (i.e., input variable assignment).
Output: Satisfying solution point $s$, or nothing.
for $i \leftarrow 1$ to $max\_steps$ do
    if $Score(s) = 0$ then
        return $s$
    $s' \leftarrow$ PickRandomlyFrom(Neighbors($s$))
    if $Score(s') < Score(s)$ then
        $s \leftarrow s'$                                   ▷ Make improving move.
    else
        if Random([0, 1)) $< e^{\frac{Score(s)-Score(s')}{TemperatureSchedule(i)}}$ then
            $s \leftarrow s'$                               ▷ Make worsening move.
```

The *evolutionary algorithm* (or genetic algorithm) is slightly different from the search strategies described above. Local search algorithms are single solution based, which means that they start from one point in the search space and move from there keeping track of only one potential solution point. Evolutionary or genetic algorithms on the other hand look for a satisfying solution in the search space by keeping track of multiple solution points at the same time, thus they are called population based or global search algorithms.

The pseudo code for the genetic algorithm is shown in Algorithm 4.8. We call the set of solution points the population. We improve the scores of individuals primarily by selecting pairs with relatively good score and mixing their features (crossover). For instance, if our solution points are bit strings, a crossover function could be taking bit assignments randomly from one parent or the other, or concatenating the first half of one parent with the second half of the other parent. The children can also go through some mutation, e.g., random bit flips. In each evolutionary step, the individual solution points in the population are getting closer to the solution.

SBST tools. The most advanced SBST based unit testing tool for C is AUSTIN [92, 93]. It uses the above described score function (4.1) and the AVM (Algorithm 4.7) as its search algorithm. Being a unit testing tool, it generates inputs (arguments) for functions. For each edge in the function's control flow graph a new search is initiated. The search is intra-procdural, meaning that only the nodes inside the function are considered for evaluating the score (approach level).

One of the most prominent example of a real-world tool using SBST is EvoSuite [62]. It is a unit test case generator for Java, using evolutionary algorithm (Algorithm 4.8). EvoSuite mutates and scores the entire test suite in one iteration, instead of trying to reach specific coverage targets with individual test cases. The test suite is scored based on the total number of covered edges. In each iteration step, the genetic algorithm treats the entire test suite as the population, and mutates it towards a bet-

Algorithm 4.7: Alternating Variable Method

```
Input: Initial input variable assignment $S = (s_o, s_1, \ldots, s_n)$.
Output: Satisfying variable assignment $S$, or nothing.
function AVM()
    for $step \leftarrow 1$ to $max\_steps$ do
        if $Score(S) = 0$ then
            return $S$
        $i \leftarrow$ Pick an input variable $i \in [0, n]$
        $progress \leftarrow$ True
        while $progress$ do
            for each $dir \in \{-1, +1\}$ do
                $progress \leftarrow$ ExploratorySearch($S, i, dir$)
                if $progress$ then
                    PatternSearch($S, i, dir$)
function ExploratorySearch($S, i, dir$)
    $S' \leftarrow (\ldots, s_i + dir, \ldots)$
    if $Score(S') < Score(S)$ then
        $S \leftarrow S'$
        return True
    else
        return False
function PatterSearch($S, i, dir$)
    $k \leftarrow 2$
    $S' \leftarrow (\ldots, s_i + dir * k, \ldots)$
    while $Score(S') < Score(S)$ do
        $S \leftarrow S'$
        $k \leftarrow 2k$
        $S' \leftarrow (\ldots, s_i + dir * k, \ldots)$
```

ter (population) score, instead of improving individual test cases. They refer to this technique as *whole test suite generation* [9].

4.4 Hybrid Techniques

4.4.1 Fuzzer - DSE Hybrids

The currently used approach to address the issues of symbolic execution is to use it in combination with fuzzing. In these hybrid systems, cheap and lightweight blind fuzzing is used to generate most test cases until fuzzing gets stuck and does not find new paths any more. This typically happens because blind mutations cannot get through complex checks. The heavyweight DSE is only used to make the fuzzer productive again by

Algorithm 4.8: Genetic Algorithm

Input: Initial population (i.e., initial set of variable assignments) *Population*.
Output: Satisfying variable assignment s, or nothing.

```
for $i \leftarrow 1$ to $max\_iterations$ do
    if $\exists s \in Population : Score(s) = 0$ then
        return $s$                    ▷ One of the individuals is a satisfying solution.
    $NewPopulation = \varnothing$
    for $i \leftarrow 1$ to $\textsc{Size}(polulation)$ do
        $x \leftarrow \textsc{SelectOneWithGoodScoreRandomly}(Population)$
        $y \leftarrow \textsc{SelectOneWithGoodScoreRandomly}(Population)$
        $child \leftarrow \textsc{CrossOver}(x, y)$
        if $\textsc{Random}() < p_{mutation}$ then
            $child \leftarrow \textsc{Mutate}(child)$        ▷ Mutation with some small probability.
        $NewPopulation \leftarrow NewPopulation \cup \{child\}$
    $Polulation \leftarrow NewPopulation$
```

getting it through the difficult checks that need the semantic insight of DSE. During the recent DARPA Cyber Grand Challenge (CGC) competition, where some of the world's best automated software security testing systems competed with each other, all systems we know of (including the winning top three) used this hybrid "fuzzing and symbolic execution" approach [121, 11, 70, 155, 151]. Most of them used AFL [180] as the fuzzer.

The idea of combining fuzzing and symbolic execution was first proposed by Majumdar et al. [104] and became a common design for automated security testing [128, 155]. By using the costly concolic execution only when it is really necessary, the overall cost effectiveness of the system can be improved. Experience showed before [151, 155, 54] that the less effective but efficient fuzzing can often perform better than more effective but costly symbolic execution.

For instance, the authors of Driller [155], compare the hybrid of AFL and their DSE engine `angr` to AFL alone and to `angr` alone. They evaluated the three configurations on 126 vulnerable DARPA Cyber Grand Challenge (CGC) applications. While AFL found 68 bugs, the DSE component only found 16. The hybrid Driller found the same 68 as AFL and an additional 9. These 9 bugs were part of the 16 that the DSE system found itself. These results suggest two things. Ones is that a hybrid system might increase efficiency (i.e., finding bugs faster), but the overall effectiveness of the whole system is the same (i.e., the bugs found by the hybrid is the sum of the bugs found by each tool separately). Second, this result also suggest that while fuzzing is less powerful than DSE, it seems to be more effective in finding the bugs that are reachable by it. This is consistent with other similar measurements [53].

As we described before, fuzzing and DSE systems make the opposite effectiveness vs. efficiency trade-off. Fuzzing is fast, but cannot break through complex conditions, while DSE is slow but more powerful. The significance of the performance cost of test generation techniques has recently been studied more formally as well by the testing community [127, 10, 8, 25, 24]. For instance, using a theoretical framework, Bohme et al. [25, 24] formally showed that even the *most effective* testing technique is *inefficient* compared to random testing if generating a test case takes relatively long. Intuitively, if it takes c times longer for concolic execution than a fuzzer to generate and run one test case in a given time frame the fuzzer can reach better coverage, simply because it can try c times more test cases. To quote the authors, while "*for 30 years we have struggled to understand how automated random testing and systematic testing seemed to be almost on par*", they finally showed that there is a limit on the efficiency of any automated systematic testing approach, beyond which random testing is *superior*. Based on their generic model they also suggest using a random-systematic hybrid [24] and show that it can perform better than either approach alone. Following these results we can make the following observation: an automated test generation system is the most efficient if every coverage goal is reached using the cheapest technique capable of reaching that goal.

4.4.2 SBST - DSE Hybrids

Search-based technique and dynamic symbolic execution systems are also used in combination. For instance, AUSTIN incorporates a degree of symbolic execution in order to generate pointer inputs, similar to CUTE. For a different tool, Baars et al. [13] developed a new approach to SBST, in which symbolic execution is used to compute a better score function for guiding SBST.

Floating point computation often pose a problem in DSE, because both the symbolic execution engine and the constraint solver need to support it. There is some work to augment DSE with search-based approaches for solving floating point computations, such as FloPSy [94] or CORAL [27].

Chapter 5

Search-based Fuzzing

Fuzzing and dynamic symbolic execution are the two main approaches currently being used for automated bug finding. Practitioners tend to prefer fuzzing because of its simplicity and scalability to large programs. Its main drawback is its lack of direction (or blindness), which makes it very hard to reach *deep bugs* that require passing many conditional branches. It is in this regard that symbolic execution excels: by employing constraint solvers, these techniques can generate inputs that can reach program points guarded by many conditions. Unlike fuzzing, symbolic execution is difficult to use on large programs, and does not scale well. Consequently, most security bugs are still found by *blind* fuzzing.

We introduce a new technique, called *search-based fuzzing* that combines some of the directionality of symbolic execution with the simplicity and scalability of fuzzers. Our design mimics the working of dynamic symbolic execution and an incomplete constraint solver, but we eliminate the need of symbolic execution by using *stochastic local search* directly on the target to find coverage increasing inputs. The resulting tool, called SBF, is able to find more bugs in a shorter period of time, when compared to state-of-the-art fuzzing and symbolic execution tools.

5.1 Search-based Fuzzing Design

To provide feedback for input generation, we instrument compare instructions in the target program. In this chapter we call compare instructions *nodes* and their two outcomes as *edges*. Note that complex if-statements will get translated into multiple nodes. We represent our control-flow graphs (CFG) in terms of these nodes and edges, as shown in Figure 5.1 for the following example:

We illustrate our approach on this example using an initial input, a single-byte vector `[0x0]`. This input takes the 0F path in Figure 5.1. Running this input through our target identification algorithm, the algorithm identifies that the true edge of node 0 (0T) has never been taken before and that it depends on the first input byte. Because of this, the edge 0T, together with its dependencies, is flagged as a local search target.

```
bool str_isprint(unsigned char *data) {
  for (; *data != '\0'; data++)
    if (!(0x1f < *data && *data < 0x7f))
      return false;
  return true;
}
```

Code Example 5.1: Printable string example

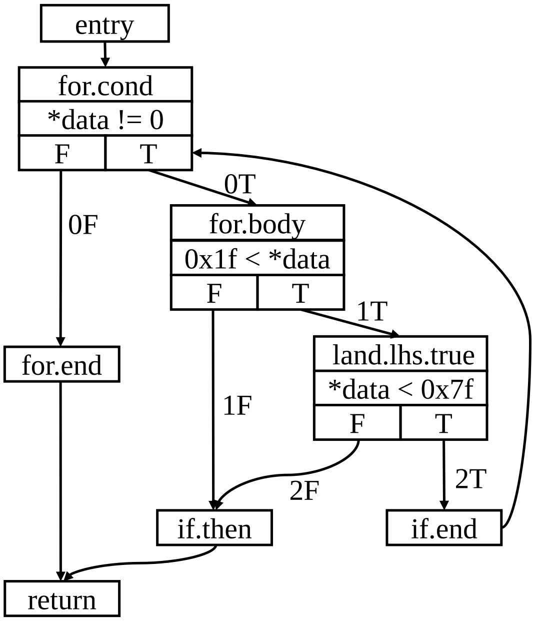


Figure 5.1: CFG of the “printable string” example.

A stochastic local search is carried out to find a value assignment to the relevant bytes that makes the execution take the targeted edge. This means that we associate a distance function with the targeted edge. Using the distance function as a feedback, the local search will quickly modify the first byte to, say, `0x1`, which will exercise the path [0T, 1F].

Having observed new coverage of the edge 1F, we next proceed to target the edge 1T with a local search. The distance value for an input assignment can be based on the Hamming-distance between the two operands of the targeted compare node. With the input `[0x1]`, the targeted comparison is `0x1f < 0x1` and the Hamming distance is 4. Reducing this distance gets us closer to negating the outcome of the comparison. When the search algorithm modifies the affecting byte, it uses this distance as a feedback to decide if a certain mutation (e.g., a bit flip) is useful or not. Through iterative improvements, the algorithm quickly finds an input, say `[0x2f]`, for which the distance is 2 and the execution takes the [0T, 1T, 2T] path. The test generation continues in this manner by targeting further uncovered edges, or even increasing the loop counts by targeting taken edges more times than before.

This high level strategy resembles that of dynamic symbolic execution systems. Say our initial input exercises the path with the condition $(data[0] \neq 0) \wedge (\texttt{0x1f} < data[0]) \wedge (data[0] < \texttt{0x7f})$. In order to exercise a different path, we want to negate the outcome of a node, say, $(data[0] < \texttt{0x7f})$. This yields the new path condition $(data[0] \neq 0) \wedge (\texttt{0x1f} < data[0]) \wedge \neg(data[0] < \texttt{0x7f})$ that is targeted next. In contrast with DSE, our algorithm does not know about this formula. It only uses the distance value which is computed by the instrumented program.

Another significant difference with symbolic execution is that we try to find a satisfying input by focusing on just the last condition. This assumes that mutating the bytes that influence the targeted edge will not invalidate earlier branch conditions. Although this assumption can often be violated, it makes our input generation incremental, something that is generally difficult in symbolic execution. The result is an input generation step that can be orders of magnitude faster, making up for lack of precision by allowing SBF to run many more iterations of its search within the same time. As shown by our experiments, this factor more than compensates for the cases where our assumption is violated.

5.1.1 Coverage Map and Distance Map

We instrument all nodes (compare instructions) in the program to compute two mappings on edges (potential outcomes): the *coverage map* $\mathcal{C}$ and the *distance map* $\mathcal{D}$. For an edge e, $\mathcal{C}[e]$ is the number of times the edge was passed. For each comparison c that was executed, $\mathcal{D}$ specifies a distance value for its last execution. For a taken edge e_{taken}, $\mathcal{D}[e_{taken}] = 0$; for a non-taken edge $e_{\neg taken}$, $\mathcal{D}[e_{\neg taken}] = d(c)$, where c is the comparison preceding the edge. The distance function $d\colon C \mapsto (0, 1]$, maps a comparison $c = x^{[n]} \bowtie y^{[n]}$ to a non-zero value, where n denotes the bit width of the comparison. We implemented two distance functions: Hamming distance $d_H(c)$, given by Equation (5.1), and arithmetic distance $d_D(c)$, given by Equation (5.2). Note that in these equations, $+/-$ is defined with overflow semantics.

$$d_H(c) = \begin{cases} 1/n \cdot max(1, H(x, y)) & \text{if } \bowtie \in \{=, \neq, \leq, \geq\} \\ 1/n \cdot max(1, H(x, y) + 1) & \text{if } \bowtie \in \{<, >\} \end{cases} \tag{5.1}$$

$$d_D(c) = \begin{cases} 2^{-n} \cdot max(1, |x - y|) & \text{if } \bowtie \in \{=, \neq, \leq, \geq\} \\ 2^{-n} \cdot max(1, |x - y| + 1) & \text{if } \bowtie \in \{<, >\} \end{cases} \tag{5.2}$$

The distance function serves as our score function for local search. It represents how close we are from taking a non-taken edge, therefore it is always larger than 0. The definitions are also slightly different in case of $\{<, >\}$ relations to avoid $(1 < 0)$ and $(0 < 0)$ having the same distance. We call the non-taken edges *touched edges*, because the execution traversed the corresponding compare instruction, but the other edge was taken. Note, that for edges that belong to non-executed nodes, we simply assign maximum distance.

5.1.2 Main Fuzzing Cycle

Algorithm 5.1 shows our high-level search algorithm. It has similarities to coverage-guided fuzzing [180, 26, 147], as well as DSE systems. This main function takes an existing set of test cases. If this set is empty, then we generate a seed test case consisting of all zeroes. The algorithm generates new test cases from existing ones. Test cases are processed in a work list (queue). Each test case in the list is fuzzed, and among the resulting mutants, those that achieve new coverage are added back to the list. The algorithm returns when the work list is exhausted. This is what we call one fuzzing cycle.

Algorithm 5.1: Main Search-based Fuzzing Cycle

```
Input: initial test suite $TestSuite$
Output: new test suite $TestSuite$ and bug triggering inputs $Crashers$
function SbfCycle($TestSuite$)
    if $TestSuite = \varnothing$ then
        $TestSuite \leftarrow \{$"00...0"$\}$
    $WorkList \leftarrow TestSuite$
    $TestSuite \leftarrow \varnothing$
    while $WorkList \neq \varnothing$ do
        $t \leftarrow$ PopOneFrom($WorkList$)
        $\mathcal{T} \leftarrow$ IdentifySearchTargets($t$)
        for each $target \in \mathcal{T}$ do
            LocalSearch($t, target$)
        for each $t' \in$ ByteMutations($t$) do
            RunAndCheck($t'$)
        $TestSuite \leftarrow TestSuite \cup \{t\}$
    return $TestSuite$
function RunAndCheck($t'$)
    $\mathcal{C}_{run}, \mathcal{D} \leftarrow$ Execute($t'$)
    if CrashHappened() then
        $Crashers \leftarrow Crashers \cup \{t'\}$
    if $\mathcal{C}_{run} \not\subset \mathcal{C}_{global}$ then
        $\mathcal{C}_{global} \leftarrow \mathcal{C}_{global} \cup \mathcal{C}_{run}$
        $WorkList \leftarrow WorkList \cup \{t'\}$
    return $\mathcal{D}$
```

To find new paths, SBF uses a more systematic, targeted strategy than existing blind coverage-guided fuzzers. For each fuzzed input SBF first establishes the set of potentially reachable new edges using a *target identification phase*, implemented by IdentifySearchTargets(); and then tries to reach them by carrying out a directed *local search* for each of those targets. The vast majority of the fuzzing time is spent in this local search phase (LocalSearch()). Optionally, an additional *blind fuzzing* step is run after this phase (the second for loop), which we will discuss more later.

The *target identification phase* identifies the edges towards which we direct our local searches. Our goal here is to negate the outcome of the nodes (comparisons) along the path taken by the initial input so as to cover previously uncovered edges. One main difference between blind fuzzing and search-based fuzzing is that we first identify these *touched* edges and direct our input mutations towards them. Trying to flip the taken branches is similar to the generational search strategy [68] of concolic execution systems.

The output of the target identification phase is the list of control dependent touched edges that if reached, would increase the global coverage. With each edge, we provide the list of input byte indices that affects the given edge. Input dependent means that the (controllable) input of the program (i.e., the attack surface) has an effect on that particular comparison (or branch). We use byte-precise data-flow tracking (Section 5.1.3) to identify input dependencies. Once the targets are identified, a local search algorithm (LocalSearch()) is run for each target. This local search aims to take the target edge by changing the affecting input bytes, using the distance map as feedback. We discuss this phase in Section 5.1.4.

Finally, in the optional *blind fuzzing* step, random blind mutations are done on each byte of the test vector, similarly to the random bit and byte flips of existing (black or gray-box) fuzzers. The ByteMutations() function returns a set of mutations of the input, where only a single byte is modified in each mutant. This phase is optional, as its primary purpose is not to find new paths, but to trigger bugs along the original path triggered by the input. For instance, considering the single path program `char buf[10]; buf[input]=0;`, we might cover the path with input 5, but the bug is only triggered with input outside of the $[0, 9]$ range.

With each mutation carried out during a local search or during the blind fuzzing phase, the program is run with the RunAndCheck() function, shown in Algorithm 5.1. Each time the coverage map is generated to detect new coverage and the distance map is generated to serve as a feedback for the local search.

5.1.3 Search Target Identification

The goal of the target identification phase is twofold: (1) to identify the set of nodes (and their edges) influenced by input I and (2) for each of those nodes, establish the set of input byte locations that affect the outcome of the given node. We will target the touched edges that the current input has an effect on and could increase the coverage (as defined in Section 5.1.5). We will focus each targeted local search to the input bytes that affect the targeted edge, thus reducing the search space of the local search to focus only on the bytes that matter.

The target identification algorithm uses byte-precise data-flow tracking. We assign different labels to each input byte and we propagate the labels of all memory locations and registers during execution. Our instrumentation at each comparison checks if the

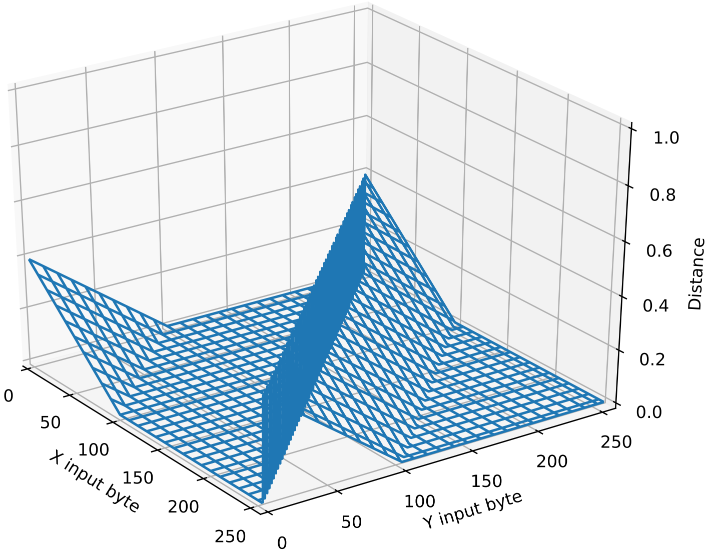


Figure 5.2: Distance function for edge $(100 < x + y)$.

arguments depend on the input, and if they do, on which input bytes exactly. After a single execution with data-flow tracking, the analysis returns a set of search targets. Each target is a pair, consisting of a yet uncovered reachable (touched) edge, and the corresponding set of input dependencies.

5.1.4 Stochastic Local Search

Once the search targets are established by the target identification phase, we start a stochastic local search for each target. During the local search we mutate the bytes that affect the target branch, with the guidance of a distance function. For instance, the distance function for the edge corresponding to the condition (100 < `sum`) in Code Example 5.2 looks like Figure 5.2. (We are using the distance function given by Equation (5.2)).

```
void sum(byte input[2]) {
  byte x = input[0], y = input[1], sum = x + y;
  if (100 < sum && sum < 150) { /* some vulnerable code */ }
}
```

Code Example 5.2: Simple example

The search algorithm can use this distance information as a feedback to determine if a potential mutation on the input would get us closer to the solution or not. A mutation can be considered as a move from one point to another on the plot. By moving more intelligently on this distance landscape than randomly, we can find a satisfying solution point faster. Next we discuss the exact algorithms we use.

Local search algorithms

A local search algorithm is composed of the following three components:

1. a distance function (also called the score function),
2. a definition of neighbors,
3. and a search strategy.

The distance function assigns a value to a point in the search space representing how *far* it is from a potential solution. The definition of neighbors determines what points do we consider *neighboring points* in the search space, the ones reachable in a single step. Finally, the search strategy defines how the search moves from point to (neighboring) point, using the distance function as a guide.

In SBF we implemented two different distance functions, two neighbor definitions and six different search strategies. We evaluated all possible combinations of these on a variety of benchmarks. We found that different algorithms work best for different targets, yet this evaluation helped us design an algorithm that works well on a wide spectrum of targets. In this section we describe this main algorithm along with the others we tested. The SBF tool uses this main local search algorithm by default, and the experiments in the evaluation section were carried out using it as well.

The two distance functions in SBF are the ones defined in Equation (5.1) and Equation (5.2), one based on Hamming distance, and one based on the difference of the two compared operands. The neighbor definition determines the possible *moves* that the search algorithm can make in one step, in other words the possible mutations on the input byte sequence. Our two neighbors functions are *BitFlip* and *AddSub*. *BitFlip* defines the neighbors as inputs that are one Hamming distance (single bit flip) away from the current one. *AddSub* adds and subtracts powers of two to and from the bytes of the current input. The neighbors of an input vector only differ in the bytes that actually affect the targeted edge. In other words, all mutations are done solely on the bytes that the targeted edge depends on.

Next we describe the search strategies we implemented and evaluated. Our first algorithm, *Random Walk* shown in Algorithm 5.2 makes a random move in every step. It is not a *real* local search algorithm, as it does not use the distance function for guidance, only to decide whether we have found a solution so the search can stop. We will only use this algorithm as a baseline in our experiments. In the following algorithms $\mathcal{D}_I[e]$ denotes the distance of an input I from edge e. $Neighbor_n(I, l)$ is the n-th neighbor of I, where l is the list of affecting byte indices.

Perhaps the most well-known local search algorithm is the *Hill Climbing* algorithm. It is a greedy algorithm, which in each step evaluates all neighbors of the current point and moves to the one with the best distance. Doing this we can get stuck in local minima, for which the common solution is to make random moves until we can make progress again. We also follow this approach. We extend the basic hill climbing with an *eagerness probability*. This addition is shown on lines 14 and 15 of Algorithm 5.3.

Algorithm 5.2: Random Walk

Input: Initial test vector I, target edge e, and the list of byte indices $l = (i_0, i_n, \ldots, i_k)$ that e depends on.

Output: Test vector reaching e, or nothing. (Note, that implicitly RunAndCheck() also saves all input that triggers new coverage.)

```
for $i \leftarrow 1$ to $max\_steps$ do
    $I \leftarrow$ PickRandomly(Neighbors($I, l$))
    $\mathcal{D}_I \leftarrow$ RunAndCheck($I$)
    if $\mathcal{D}_I[e] = 0$ then
        return $I$
```

During the evaluation of the possible moves, if the currently evaluated neighbor improves the distance, with some probability we take that move before evaluating all possible moves. When we eagerly move when evaluating the ith neighbor, we continue by evaluating the $i+1$th neighbor of the new point.

In its limit, when the eagerness probability is 1, the algorithm evaluates the possible moves in order and immediately takes any improving one. This is especially useful when we target an equality, e.g., `if (input == magic_number)`. Suppose we use the *BitFlip* neighbor definition, and Hamming distance as the distance function. In this case, the local search algorithm will go through each bit of `input` one by one, and flip them if that particular bit is different in `magic_number`. This algorithm is guaranteed to find the right input in maximum as many steps as the number of bits in `input`. When eagerness probability is one we will refer this algorithm as *Eager*, otherwise we call it Hill Climbing.

Our next algorithm is Simulated Annealing and its special case, Markov Chain Monte Carlo (MCMC) sampling. This strategy does not evaluate all options in every step, but picks a random neighbor, then decide whether to move there or not based on the neighbor's distance value. When the distance is better (smaller), we always make the move. However, to be able to escape from local minima, we also accept non-improving moves with some probability. This probability is determined by the Metropolis-Hastings acceptance criteria, shown on line 11 of Algorithm 5.4.

Originally, MCMC is a method for sampling from probability distributions, so that we take samples from regions with higher probability density more often than regions with lower probability. However, the technique can be directly applied as a local search algorithm, sampling from a search space according to a distance function. This type of stochastic local search, has been used by others as well, for instance, by STOKE [139] for superoptimization.

Simulated Annealing is another adaptation of the Metropolis-Hastings acceptance criteria, where additionally to the MCMC algorithm we also lower the acceptance probability with time. It adds a T *temperature* factor to the algorithm, which is reduced according to a γ cooling factor, as shown on line 13. In the extreme case when

Algorithm 5.3: Hill Climbing with Eagerness Probability Extension

Input: Initial test vector I, target edge e, list of byte indices $l = (i_0, i_n, \ldots, i_k)$ that e depends on, and the probability of taking non-worsening move eagerly (without evaluating all neighbors) p_{eager}.

Output: Test vector reaching e, or nothing. (Note, that implicitly RunAndCheck() also saves all input that triggers new coverage.)

```
for $i \leftarrow 1$ to $max\_steps$ do
    $progress \leftarrow$ True
    while $progress$ do
        $progress \leftarrow$ False
        $N_{best} \leftarrow I$
        for $n \leftarrow 1$ to $number\_of\_neighbors$ do
            $N \leftarrow Neighbor_n(I, l)$
            $\mathcal{D}_N \leftarrow$ RunAndCheck($N$)
            if $\mathcal{D}_N[e] = 0$ then
                return $N$                                        ▷ Found a solution.
            if $\mathcal{D}_N[e] < D_{N_{best}}[e]$ then
                $progress \leftarrow$ True
                $N_{best} \leftarrow N$
                if Random($[0, 1)$) $< p_{eager}$ then
                    $I \leftarrow N$                              ▷ Make move eagerly.
        $I \leftarrow N_{best}$                                   ▷ Make the best move.
    $I \leftarrow$ PickRandomly(Neighbors($I, l$))
```

$\gamma = 1$, the two versions of the algorithm is equivalent. In this special case, we will refer to this algorithm as *MCMC*, otherwise as *Simulated Annealing*.

Algorithm 5.4: MCMC and Simulated Annealing

Input: I test vector, e target edge, $l = (i_0, i_n, \ldots, i_k)$ list of byte indices e depends on, β accept probability factor, γ cooling factor

Output: Test vector reaching e, or nothing. (Note, that implicitly RunAndCheck() also saves all input that triggers new coverage.)

```
$T_0 \leftarrow 1.0$
for $i \leftarrow 1$ to $max\_steps$ do
    $N \leftarrow$ PickRandomly(Neighbors($I, l$))
    $\mathcal{D}_N \leftarrow$ RunAndCheck($N$)
    if $\mathcal{D}_N[e] = 0$ then
        return $N$                                        ▷ Found a solution.
    $\Delta \leftarrow \mathcal{D}_N[e] - \mathcal{D}_I[e])$
    if $\Delta < 0$ then
        $I \leftarrow N$                                  ▷ Improvement found.
    else
        if Random($[0, 1)$) $< e^{\frac{-\Delta}{\beta \cdot T_i}}$ then
            $I \leftarrow N$                              ▷ Acceptance criteria passed.
    $T_{i+1} \leftarrow \gamma \cdot T_i$
```

Our final algorithm, which we designed particularly for SBF, is the combination of the Eager and MCMC algorithms. We run the the above described Eager (Algorithm 5.3 with $p_{eagerness} = 1$) until that strategy stops making progress (the first time we get stuck), then we switch to MCMC. As the following benchmarks confirm, this helps in finding solutions for easy targets, such as magic values as quickly as possible, while also allows solving hard targets. We refer to this algorithm as *EagerMCMC*.

In order to evaluate the described algorithms, we ran all possible combinations of the two distance functions, two neighbors definitions, and six search strategies, with different parameters, on a set of benchmarks. Considering the different parameters (e.g., for β or $p_{eagerness}$), this adds up to 236 different configurations of local search algorithms. We tried these configurations on ten small benchmarks. Each benchmark does some computation on the input and compare the result of that computation to some constant (i.e., equals/larger than). The computations include calculating different checksums, such as modular sum, Adler, and Fletcher. Others convert the input string to an integer or a float number, or implement a polynomial function and some other simple functions. The goal of each search in the benchmark is to find an input that takes to true edge of the comparison.

We ran all configurations on all benchmarks 1000 times, with $max\_steps = 100000$, and measured how many times the search found a solution. Table 5.1 summarizes the results. We do not list all 236 configurations, only the best configuration for the

Search strategy	**Distance**	**Neighbors**	**Success rate**
EagerMCMC ($\beta = 0.2$)	Hamming	AddSub	94.81%
MCMC ($\beta = 0.2$)	Hamming	AddSub	94.54%
SimulatedAnnealing ($\beta = 0.2, \gamma = 0.999$)	Hamming	AddSub	92.89%
HillClimbing ($p_{eagerness} = 0.1$)	Difference	AddSub	89.08%
Eager	Difference	AddSub	80.23%
RandomWalk	Difference	BitFlip	10.27%

Table 5.1: The best configuration of each algorithm and its average success rate on our benchmark.

six different algorithms we described earlier. We indicate the distance and neighbors function used in the configuration, and the parameters that were used for the search strategy.

The combined EagerMCMC algorithm, using Hamming distance and AddSub neighbors definition had the highest success rate. MCMC combined with the Eager algorithm does significantly better only on the easiest targets (e.g., input equals to a constant), and similarly on the other ones, therefore the overall success rate is just marginally better. We also found that Simulated Annealing results were always worse than MCMC. This is why Simulated Annealing performed the best when its γ parameter was closes to MCMC's. Hill Climbing did worse than MCMC in general, but interestingly the best performing configuration was when the eagerness probability was 0.1. This means that our eagerness extension improves on the standard algorithmm, where this probability is zero or not used. As mentioned before, in its other extreme, when the eagerness is 1, we call the algorithm just Eager. Again, this only works well for easy targets, such as comparing the input to a magic value. Unsurprisingly, the blind Random Walk strategy produced the worst results. It still reached ~10% success rate due to some of the simple benchmarks, but in case of more complex ones, such as the Adler checksum, the random mutation strategy never found a solution in the given steps limit.

It is also interesting that *AddSub* gave slightly better results even in combination with the Hamming based distance measure. One might expect that the single bit flipping mutation strategy would work better with Hamming distance. With *AddSub* we also systematically flip bits, but the set of neighbors are larger than for *BitFlip* (15/byte vs. 8/byte). Having a larger set of neighbors turned out to be beneficial.

Targeting paths vs. targeting edges

There are two ways we can direct a search towards an uncovered edge. We can target a specific path prefix leading to the edge we want to reach, or we can disregard the path prefix and only focus on the comparison whose outcome we want to negate. We call the first *path targeting*, and the other *edge targeting*. In case of path targeting, the distance feedback for the local search need to take into account all nodes along the

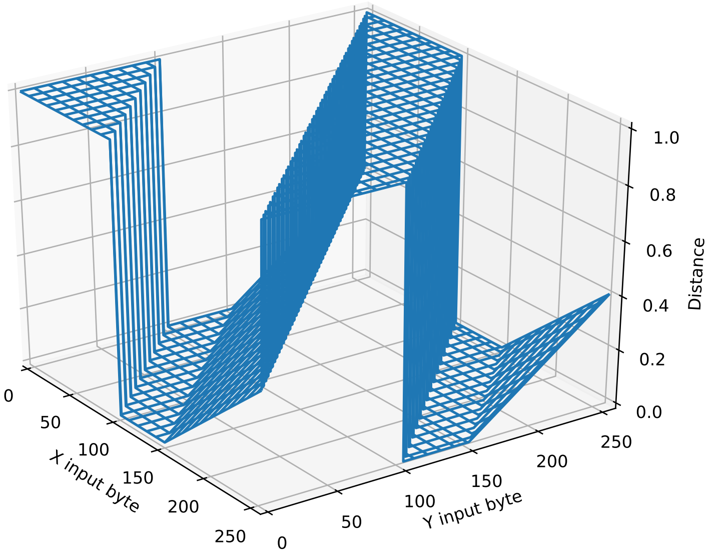


Figure 5.3: Distance function for edge $(x + y < 150)$.

path prefix. We can define such *path distance* as the sum of the node distances along the targeted path: $\sum_{\forall e \in Path} \mathcal{D}[e]$. While in case of edge targeting we simply use $\mathcal{D}[e]$ as the feedback.

Recall that we only target edges that are *touched* by the path of the currently mutated input. This means that if we target an edge, the current input already exercises the path prefix leading to it, so all node distances along that path prefix will be zero. In other words, initially the path distance and the edge distance will be the same. We rely on this fact to make an optimization and only take into account the distance for the targeted edge, not considering its path prefix, and the dependencies of all prior nodes. This, however, is an opportunistic optimization, because when we start mutating the input, we can easily get off the original path prefix and we might not even reach the targeted node at all.

As an example, consider the distance function in Figure 5.3, which represents the distance for edge corresponding to the condition `sum` < 150 in Code Example 5.2. This node is guarded by the condition $100 <$ `sum`. If this condition is not satisfied, we do not reach the second node. These are the points with non-zero distance in Figure 5.2 on page 81. Recall, that for unreached edges we assign maximum distance, which creates a plateau in the distance landscape, which hinders guiding the local search. However, plateaus only cause a problem if we start the search from them. In our example we are guaranteed to start a search from a point outside the plateau areas, because we target this edge from an input that already passes the first comparison. In other words we start from one of the zero-distance points in Figure 5.2, satisfying the $100 < x + y$ condition.

Our optimization of targeting edges as opposed to specific paths reduces the search space significantly and works well when the preceding conditions along the path prefix are independent from the target condition (when they depend on different parts of the input). The example above we showed that it can also work well when we have to satisfy consecutive conditions on the same input bytes. The optimization does not work well when some mutation breaks a previous condition along the path that depends on some of the same input bytes as the targeted edge. In general we found targeting edges and ignoring the path leading to an edge a better strategy than targeting specific paths. Primarily because by directing the search towards an edge, but not forcing it to take a specific path, we make the search less focused. This allows the (cheap) mutations to discover new paths and edges different from the actual target. This is a middle ground strategy between coverage-guided fuzzing, which does not target anything, and between DSE, which targets a specific path. We found this design point in the middle a sweat spot.

5.1.5 Coverage Metric

The SBF design focuses on increasing branch coverage: it targets edges and uses edges as a coverage metric. The coverage metric determines whether an input (mutation) is new coverage, and should be added to the test suite. Specifically, we use counted edge coverage, which means that we consider an input new, if it covers a new edge, or covers a previously covered edge more times than before. This is a similar metric that was introduced by AFL, and used by existing coverage-guided fuzzers, however there is an important difference. Recall that SBF deals with the edges of comparison nodes, while existing tools [180, 147] deal with the edges of the control-flow graph. Tools measuring CFG-edges register jumps from one basic block to another, by instrumenting jump instruction and/or basic blocks. The edge coverage metric of SBF registers the outcome of individual comparisons, by instrumenting compare instructions only.

This has multiple benefits. First of all, we only have to instrument compare instructions for maintaining both the coverage map (to target and detect new coverage), and the distance map (to guide mutation). Second, by instrumenting individual compare instructions we can effectively attack the problem of complex branch conditions. When a CFG edge is only taken if a complex branch condition is satisfied (e.g., `if (a && b && c)`), a fuzzer relying on CFG edge coverage can only detect when all sub-conditions become true. In contrast, using SBF's metric, we can detect incremental improvements, e.g., when one or two of the sub-conditions get satisfied. This more fine grained coverage feedback helps making progress, even without SBF's targeted search strategy, but also using just blind mutation.

5.1.6 Test Suite Inflation and Deflation

Relying on (counted) edge coverage have certain limitations. To demonstrate the issues, consider the example in Code Example 5.3, which is taken from a KLEE tutorial [105]. The function takes a string, where each character represents taking a step in a maze. To win the game the character sequence must lead through the maze reaching the goal shown as #, i.e., the string ddddrrrruulluurrrrddddrruuuu will reach the “You win!” edge. Let us consider what inputs can be generated using different coverage metrics. The four interesting edges (other than the winning one) are the ones corresponding to taking the four different directions. With (uncounted) edge coverage, the test generation stops after generating the inputs u, d, l, and r, because with these we already covered taking all four directions. With *counted* edge coverage we can also find dd, ddd and dddd, because covering the same edge more times constitutes as new coverage. However, it will get stuck at that point and will not find ddddr, because it already generated r, in other words, we already covered “moving right once”.

```
int walk_maze(char *steps) {
  char maze[H][W] = { "+-+---+---+",
                      "| |     |#|",
                      "| | --+ | |",
                      "| |   | | |",
                      "| +-- | | |",
                      "|     |   |",
                      "+-----+---+" };
  int x = 1; int y = 1; // Player position.
  for (int i = 0; i < MAX_STEPS; i++) {
    switch (steps[i]) {
      case 'u': y--; break;
      case 'd': y++; break;
      case 'l': x--; break;
      case 'r': x++; break;
      default: printf("Bad step!"); return 1; }
    if (maze[y][x] == '#') {
      printf("You win!"); return 0; }
    if (maze[y][x] != ' ') {
      printf("You lose."); return 1; } }
  return 1; }
```

Code Example 5.3: Maze example

The problem is that in order to generate that input, we would need to take a previously covered edge, but from a different context. One potential solution for this problem would be to differentiate between all paths or path prefixes. This, however, would lead to a test suite blow up due to the practically infinite number of paths in any non-trivial program. We solve this problem differently and eliminate the limitation of counted edge coverage while avoiding using a path coverage metric.

We introduce an outer loop shown in Algorithm 5.5 around the main fuzzing cycle, SbfCycle() (Algorithm 5.1). Every time the work list gets exhausted, we take the generated test suite, minimize it, randomize it, and restart the main fuzzing cycle with

Algorithm 5.5: Test Suite Inflation & Deflation

```
Input: initial test suite $TestSuite$
Output: new test suite $TestSuite$
while not stopped do
    $InflatedTS \leftarrow$ SbfCycle($TestSuite$)
    $DeflatedTS \leftarrow$ SetCover($InflatedTS$)
    $TestSuite \leftarrow$ Randomize($DeflatedTS$)
    $\mathcal{C}_{global} \leftarrow \varnothing$                ▷ Reset global coverage map.
return $TestSuite$
```

the global coverage reset to empty. Resetting the global coverage allows subsequent cycles to rediscover the same edges, but from a different context, through the mutation of a different input. Preferably, we want to re-take edges from a deeper point of the execution than before. This is why we minimize the test suite after each cycle using a naïve approximating set covering algorithm. We eliminate redundant test cases that cover edges that are also covered by inputs that cover more edges. In other words, we prioritize the test cases that exercise longer paths. This way the next cycle can start from a better input, and by rediscovering some edges from deeper in the program, newer states might be reached. We do not want however to always start fuzzing the input with the deepest path, therefore before each cycle we also randomize the order of the test cases. This makes sure that edges are rediscovered from a variety of contexts in each fuzzing cycle.

We call this technique *test suite inflation & deflation*, because in each round, the main fuzzing cycle inflates the test suite and the outer loop deflates it. The inflation is because, due to the cleared global coverage, a number of new inputs will be added to the test suite. Even when these are redundant from the edge coverage perspective, they exercise new paths, which enables reaching completely new edges. Typically, after the first cycle, each inflation approximately doubles the test suite size and each deflation halves it. With this strategy, we avoid test suite blow up, while we monotonically increase the global coverage.

In case of the running example, after the first cycle that gets stuck with the input `ddddr`, the second cycle will likely start by fuzzing this input, take the `r` edge from it and proceed in a similar fashion until everything is covered. Indeed, SBF reaches 100% coverage in this example in just a few microseconds.

5.2 Implementation

The SBF implementation has two parts, the SBF compiler and the SBF library. The SBF compiler extends the LLVM/Clang compiler with a plugin that performs the instrumentation. The SBF library contains the implementation of the test generation algorithms described so far. Like LibFuzzer, SBF links to the instrumented target with the SBF library in order to create a single executable. As a result, fuzzing can execute within a single process. Also like LibFuzzer, programs to be fuzzed need to implement an entry function with the following interface: `void run(char *input, size_t size)`. The fuzzer runs one test case by calling this function with the generated input.

5.2.1 Instrumentation for the Coverage and Distance Maps

The main SBF instrumentation transforms the target program to maintain the distance map and the coverage map. It is implemented as an LLVM compiler pass, working on the intermediate representation (IR). This pass only modifies compare instructions. While the LLVM IR contains a `switch` instruction as well, we lower them to compares before carrying out the instrumentation.

After each compare instruction, we add a short code snippet that updates the distance and the coverage map. The distance map contains the distance values and the coverage map contains the hit counts for each edge. Each edge is assigned a unique ID. The distance and coverage maps are arrays, reserving one byte for each edge, indexed by the edge ID. The instrumentation after each comparison computes and stores the distance value of the non-taken edge, while it sets the distance to zero and increments the coverage counter for the taken edge.

5.2.2 Instrumentation to Identify Search Targets

The search target identification relies on a different instrumentation that used to construct a data structure representing the search targets. This is a list of pairs consisting of the ID of the target edge and the list of byte indices that this edge depends on. We rely on dynamic data-flow tracking (DFT) to get this information. LLVM already includes a DFT library, called DataFlowSanitizer, which we use to build our target identification on. DataFlowSanitizer is able to propagate multiple (taint) labels. We assign a separate label to each input byte index. After each comparison instruction, we check the labels associated with the arguments of the comparison. If there are associated labels, and the edge is not covered yet (as many times as we could from the current context), we add the corresponding edge ID to the list of targets, along with the byte indices represented by the labels.

Note that DFT is needed only during the target identification phase. Since DFT has a significant runtime performance cost, we would like to avoid it while doing the local searches. One approach for this is to create two binaries, one with DFT, and one without. Synchronizing and communicating between two processes however would add significant complexity and overhead. We therefore devised an alternative approach: we still create two versions of the targeted code, but link them into the same binary by renaming functions and other globals in one of them. Function versions that include DFT are prefixed with `dft_`. With this naming scheme, SBF calls `run()` during local search runs, and calls `dft_run()` for target identification.

5.2.3 Compiler Optimizations

We leverage compiler optimizations to create more edges, which enables finer granularity measurement of coverage. In particular, *Loop unrolling* and *function inlining* have the most potential for increasing the number of edges. Consider the following loop, for example:

```
for (int i=0, i<3; ++i)
  if (!(isprint(input[i])))
    return 0;
return 1;
```

When compiling the target with SBF the above gets unrolled as shown below.

```
if (!(isprint(input[0]))) return 0;
if (!(isprint(input[1]))) return 0;
if (!(isprint(input[2]))) return 0;
return 1;
```

In the original program there was only one input dependent compare node, but in the transformed one there are three. In the unrolled loop we have more edges and are able to differentiate between different execution paths.

Function inlining helps in a similar manner. We can differentiate between the edges of the inlined function when they are called in different contexts, as there will be separate code for each context. Parser code often uses standard library functions for character or string comparisons, such as `isascii`, `isspace`, or `strcmp`, and `memcmp`. We also make sure that all these small C library functions are inlinied. We collected the implementation of these typically single line functions from the `musl` C library and make sure that they get inlined when we compile a target with the SBF compiler.

5.2.4 The SBF Library

The SBF library contains the implementation of the SBF algorithms described in this paper, including the fuzzer cycle, the local search algorithms, the coverage check, etc., and also the `main` function. It is written from scratch, without using any external libraries and consists of about 6 KSLOC of C++.

Crashing signals are handled and state saving and restoring is also implemented. The state of the fuzzer consists of the last minimized test suite, the global coverage map and the statistics counters. These three are serialized when the process is stopped and restored when restarted. For more precise bug detection, optimally ASAN [146] or other LLVM sanitizer insturmentations can be used.

The hottest function in the code is the one comparing and updating the global coverage ($\mathcal{C}_{global}$ with $\mathcal{C}_{run}$) after each run. We implemented an optimized version of this function using the Intel AVX2 instruction set to process the arrays 32 bytes at a time, which resulted in a around 2× speedup.

5.3 Evaluation

We evaluate SBF by comparing it to three other test generation tools, namely to AFL [180], LibFuzzer [147], and KLEE [31]. AFL was the first fuzzer that demonstrated that coverage-guided fuzzing can be effective. LibFuzzer implements the same technique as AFL, but instead of forking a new process for running each test case, it does *in-process* fuzzing, by linking the fuzzer with the target library. As described above, we also follow this design with SBF. KLEE is the the most well known open source dynamic symbolic execution engine. We are first to provide an in-depth comparison of AFL, LibFuzzer and KLEE with each other on a range of programs.

We are most interested in the tools' effectiveness in finding bugs. Unfortunately, directly measuring the number of bugs found by a test generation tool in various programs is not a good metric, because of multiple reasons. First, bugs are relatively sparse in programs, which means that they provide a very low resolution metric. Second, it is hard to measure the number of unique bugs found by a fuzzer. Fuzzing tools find crashes, but the number of crashes found does not equal the number of bugs found. To precisely establish the number of unique bugs given a set of crashes, one needs to do thorough root cause analysis, which typically can only be done by manual inspection as programmatic heuristics are not reliable.

Bugs, however, are just erroneous program states, and to be able to trigger them, the tool needs to be able to reach that state in the first place. Therefore, by far the most important aspect of a bug finding tool is its ability to discover new program states, in other words its ability to increase coverage. This is why we focus our evaluation on measuring the coverage reached by the tools, instead of the number of bugs found. Note, that there can be a difference between two tools bug triggering ability along the paths that we can already cover with existing inputs. For instance, given the same test suite, one tool might be able to trigger more bugs along the exercised paths than the other. With SBF, however, we did not make any contributions in this aspect compared to other fuzzers. Both the state-of-the-art fuzzers and SBF uses the same random blind mutations to trigger the bugs, and potentially sanitizers (e.g., ASAN [146]) to detect subtle errors more precisely along paths that we can cover. This means that SBF finds more bugs iff it covers more execution states than the competing tools.

Nevertheless, we evaluate SBF's bug finding ability directly as well and compared it with the other tools. To deal with the sparsity problem and the unique bug identification problem, we rely on LAVA [54], a recently published tool that automatically inserts vulnerabilities into programs. LAVA can insert a high number of bugs in programs, each identified by a unique ID, which enables us to do a rigorous evaluation. Our results show that SBF can reach higher coverage and find more bugs under minutes, than competing tools under hours.

We also evaluated the contribution made by each of the three components of SBF to its coverage increasing ability. To this end we tested SBF in the four configurations summarized in Table 5.2. The first, $\mathrm{CGF}_{\mathrm{base}}$, is the most basic coverage-guided fuzzing algorithm, to which we add the SBF features one by one. For $\mathrm{CGF}_{\mathrm{base}}$, we run Algo-

	Inflation & Deflation	**Targeting & mutation focus**	**Local Search**
CGF_{base}			
$\text{SBF}_{\text{blind}}$	✓		
$\text{SBF}_{\text{targeted}}$	✓	✓	
SBF_{full}	✓	✓	✓

Table 5.2: Four configurations of SBF indicating the enabled features.

rithm 5.1 with the search target identification and local search switched off, keeping only the random byte mutation phase. We repeat the main cycle without the test suite deflation and coverage reset in the outer loop. This means that we simply keep going through each test case in the test suite, and for each test case, we go through each byte and randomly mutate them.

In the next configuration, called $\text{SBF}_{\text{blind}}$, we enable the test suite inflation/deflation in the outer loop. With, $\text{SBF}_{\text{targeted}}$, we also add target identification. Here the local search function is also on, but it uses the RandomWalk algorithm, as defined in Algorithm 5.2. This means that the distance function feedback is not used at all. Essentially, we do the same random byte mutations, as in case of $\text{SBF}_{\text{blind}}$, but this time it is focused only to the bytes on which the targeted edge depends on. Finally, SBF_{full} is where all three features are switched on. In short, while SBF tells *what* bytes to mutate and *how*, $\text{SBF}_{\text{targeted}}$ only tells the *what*, while $\text{SBF}_{\text{blind}}$ tells neither.

5.3.1 Coverage Increasing Ability

We measured program coverage growth in time as SBF and the competing tools generated test cases. We picked four programs among the targets of the Google's OSS-Fuzz project [3]. Two that takes textual inputs: `libxml` and `pcre`, and two that parses binary formats: `libpng` and `libjpeg`. We ran the four tools on the four different targets for 12 hours each. We repeated all experiments 4 times to also measure the variability of these randomized algorithms.

As SBF and LibFuzzer are in-process fuzzers, they require the definition of an entry function, i.e., the `run()` function described in Section 5.2, that calls the tested functionality. This typically means invoking the main parser function on the provided input buffer. Such functions were already available for our targets due to OSS-Fuzz. For testing with AFL and KLEE, we also linked a thin `main()` function to the targets. This takes a file as a command line argument, reads its contents and calls `run()` with it.

We used AFL and LibFuzzer with their default options and KLEE with the suggested set of flags provided on the tool's website, which were used for the `coreutils` experiments as well in their paper [31]. While the `coreutils` experiment used different symbolic input sizes for different targets, we set the symbolic input size to 64 bytes for all experiments. To match KLEE's input size, we set the maximum input size to

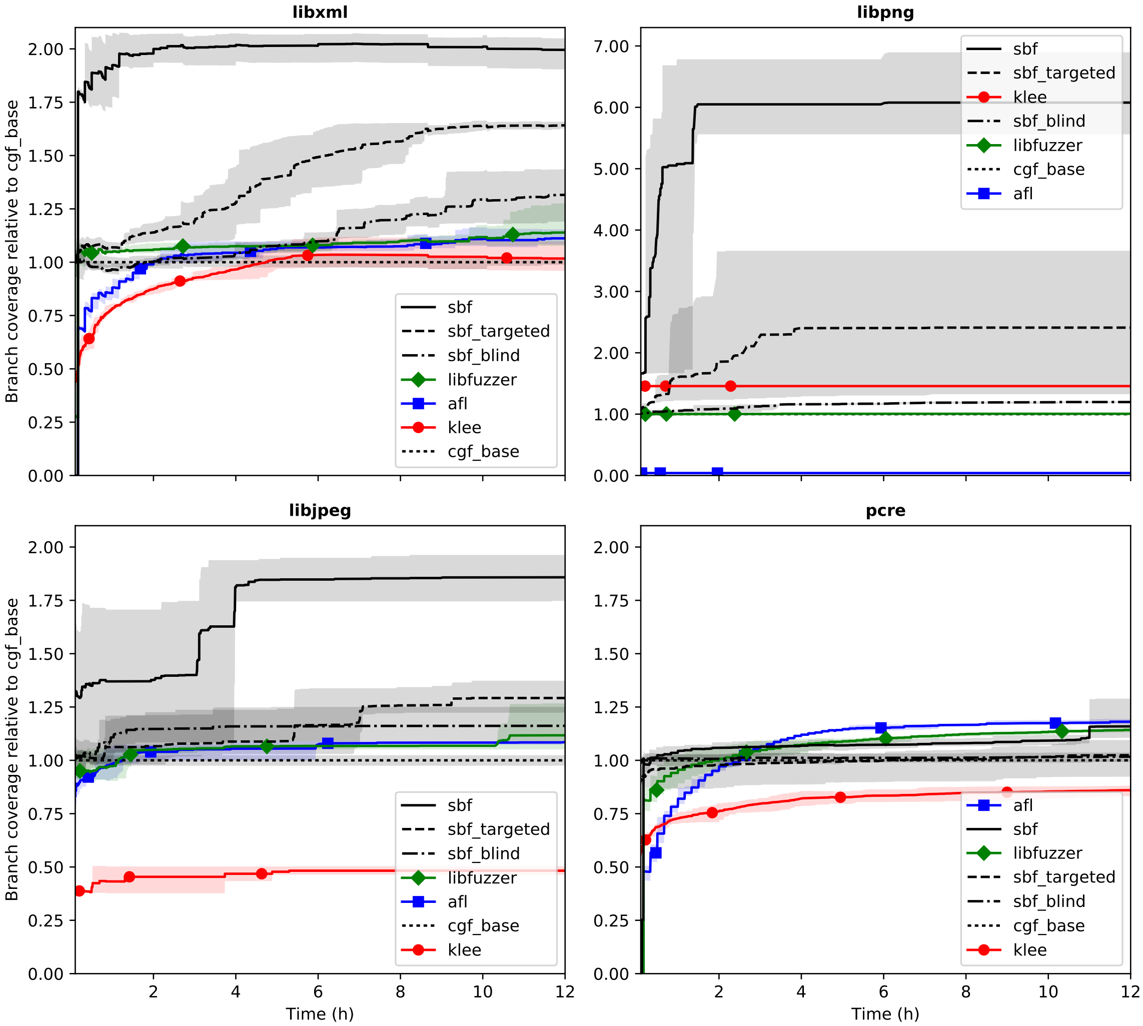


Figure 5.4: Coverage growth on four benchmarks, relative to base coverage guided fuzzing algorithm CGF$_{\text{base}}$.

64 for the other three tools as well. Further, as KLEE cannot be provided with seed inputs, we started the test generation with an empty test suite with all tools, to make the comparison fair.

We measured the coverage using `gcov`. We modified all competing tools to produce the same timing log as SBF, capturing the exact time of the generation of each new test case. We used this log to replay the created test cases on the `gcov` instrumented program, measuring the branch and line coverage as a function of time.

Figure 5.4 shows branch coverage for each tool normalized to the coverage reached by CGF_{base} at the given time. The x-axis shows the time the test generation tools ran, starting from 5 minutes to 12 hours. During the first five minutes the relative behavior is not stabilized yet, therefore it is not shown. On the y-axis 1 represents the coverage reached by CGF_{base}. The lines represent the mean of the four executions, while the shadows around the lines mark the area between the minimum and maximum values, to show variability.

In `libxml2` (a library for parsing XML documents) SBF obtains approximately twice as much coverage as the other tools. Note that it reaches significantly higher coverage in the first few minutes than what is achieved in 12 hours by the other tools. The three competing tools reach similar levels of coverage as CGF_{base}, with AFL and KLEE taking several hours to get there. $\text{SBF}_{\text{blind}}$, with only the inflation/deflation switched on, already outperforms all three competing tools. $\text{SBF}_{\text{targeted}}$, switching on focused mutation, further improves coverage by 50% after 12 hours. Finally with full-featured SBF, we can see the effect of the guided local search, with which we reach high coverage in a just few minutes. The XML format has a complex grammar, but SBF creates XML files with a wide variety of XML tokens and keywords in them.

On `libpng`, which is the official reference library for the Portable Network Graphics (PNG) format, CGF_{base}, AFL, LibFuzzer, and KLEE saturate after just a few minutes. They stop making progress at slightly different coverage levels. AFL gets stuck covering very little, LibFuzzer reaches the exact same coverage as CGF_{base}, while KLEE saturates somewhat higher. On the other hand, SBF reaches 6 times more coverage than the base algorithm and LibFuzzer, after less than two hours. The `libpng` library is a relatively hard target, as PNG is a complex file format. Still, SBF generates the proper header, with the necessary signatures and also internal "chunks" identified by different four letter codes, without any initial seed input. Considering the contribution of the SBF features, switching on the targeted, focused mutation already improves a lot on the effectiveness. As in `libxml`, the graph clearly shows that adding the feedback of the stochastic local search has a huge impact on the effectiveness.

The `libjpeg` benchmark implements JPEG image handling functions, including image encoding, decoding, transforming, etc. The benchmark program runs the JPEG decode/decompression function on the input. The two coverage-guided fuzzers perform very similarly to CGF_{base}, while KLEE saturates with half of the coverage reached by the fuzzers. SBF reaches significantly higher coverage than the competing fuzzers in a few minutes, and around 1.8× as much coverage in a few hours.

All tools perform very similarly on the last benchmark, the Perl Compatible Regular

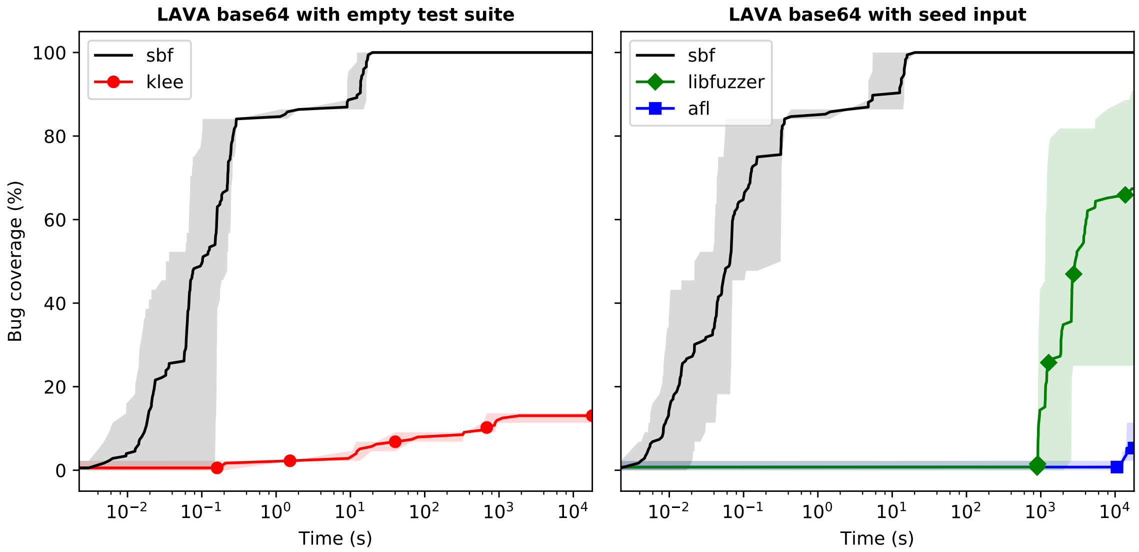


Figure 5.5: Bugs found in the LAVA base64 benchmark. Time is in logarithmic scale.

Expressions (v2) library, or PCRE2 in short. SBF leads during the first hours, AFL and LibFuzzer catch up in the long run and they saturate at around the same level as SBF. KLEE progresses slower and covers less than the other tools. We believe the reason there is not much difference between the performance of the three fuzzers on this benchmark, is because the target is easily discoverable even with completely blind and random fuzzing. The tokens of a regular expression are typically single characters, which means that it is easy to generate a wide variety of expressions, just through random byte mutations.

5.3.2 Bug Finding Ability

Measuring the number of bugs found in programs with only a few bugs in them is not statistically robust. To directly evaluate the bug finding ability of tools, we need targets with a large numbers of bugs in them. LAVA [54] is a tool that enables such evaluation by automatically adding many realistic vulnerabilities to existing code. In order to evaluate SBF's and the other tool's bug finding power, we used one of the LAVA benchmarks, namely the LAVA modified `base64` utility that has 44 injected vulnerabilities in it. We ran all four tools on this target for 5 hours (same as the experiment in the LAVA paper [54]), and as before, we repeated the experiment four times, in order to get an average performance.

In order to make the `base64` utility testable with SBF and LibFuzzer, we replaced the original `main` of the program (that parses command line arguments) with a `run()` entry function that calls the Base64 decode function on the provided input buffer. With AFL and KLEE we tested the same code, linked with an additional `main()` function to read the input from a file and pass it to `run()`.

We ran two experiments, one where we ran the tools with an empty initial test suite, and one where a seed file was provided. The results are shown in Figure 5.5. We plot the percentage of the found bugs growing with time. Note that the time axis is in logarithmic scale. Without a seed input, only SBF and KLEE found bugs. SBF found all bugs in around 15 seconds in each run, while KLEE found 5 to 6 bugs each time, in 19 minutes on the average.

In the second experiment we provided the fuzzers with an initial Base64 encoded seed input (the one provided by the benchmark). KLEE is not shown on this plot as it does not accept seed inputs. This time both AFL and LibFuzzer were able to find bugs. AFL found 5 bugs in one of the runs, but no more than one in the other runs. On average, this is 2 bugs in 4 hours. LibFuzzer did better with an average of 22 bugs in 3 hours and 15 minutes. SBF performed similarly with or without a seed input, and found all injected bug in a matter of seconds.

The reason why SBF is so much more effective in finding bugs is that LAVA introduces a new edge for each vulnerability it injects. SBF targets each of these, and the directed local search algorithm can modify the relevant input bytes to trigger the targeted bug very quickly. In contrast, blind fuzzers do not target any particular edge and they carry out their mutations randomly, therefore it is only a matter of luck and time whether bugs are found.

5.4 Comparison to Related Work

Fuzzing. Black-box fuzzers [112, 174, 135, 189] run the target with randomly mutated or generated inputs without any feedback. If we already have a test suite with high coverage, these fuzzers can trigger bugs along the paths covered by this test suite, but they are not effective in increasing coverage. For this reason, modern fuzzers such as AFL [180] and LibFuzzer [147] are coverage-guided: whenever a mutant happens to cover a new path, it is selected for additional mutation. Using coverage to provide feedback in this manner, these fuzzers are able to increase coverage. They may even be able to operate without any initial test suite. Nevertheless, mutations themselves are carried out blindly. SBF, in contrast, incorporates a *directed* approach based on light-weight instrumentation to decide *what* to mutate, and *how* to mutate.

AFLFast [26] is a recent extension of AFL that finds new paths faster by applying a heuristic on which files to fuzz more. The intuition behind its strategy is that there is a better chance of reaching new edges from inputs that exercise rare paths, so they prioritize fuzzing those inputs more. This is a useful heuristic when the fuzzer has no information on what exactly are the reachable (touched) new edges from an input's path. With SBF, due to the target identification phase, we have this information, and we can avoid fuzzing inputs that do not touch any uncovered edge. Our targeting strategy inherently applies AFLFast's prioritization even more precisely, as we only fuzz inputs that have the potential to reach a new edge. The more reachable edges there are, the more we fuzz, as the number of identified edges determines the number of local searches.

Taint-guided fuzzers [64, 170, 16, 102] use data-flow tracking to decide which parts of a given input to mutate more than others. These tools use taint-tracking to identify what parts of the input flow into potentially dangerous operations, such as library function calls (e.g., `malloc` or `strcpy`), and fuzz those parts of the input more. This can help in triggering a bug earlier along a path covered by an existing test suite, but does not help in finding new paths. More importantly, they do not address the problem of *how* to mutate.

Dynamic Symbolic Execution. Dynamic symbolic execution [67, 145, 31, 36, 32], also called *white-box fuzzing* [68], is a systematic testing approach. It generates inputs in two steps. First, the program is run symbolically in order to extract a path condition formula. Then, as a second step, a constraint solver (SMT solver) is used to find input assignments satisfying formulas representing alternative paths. SBF achieves some of the directionality of DSE while avoiding the (costly and complex) symbolic execution step. Recently, a stochastic local search (SLS) based constraint solver [63] was shown to be competitive with commonly used (systematic) constraint solvers [48, 15]. The SLS based SMT solver of Fröhlich et al. [63] even outperformed the state-of-the-art CDCL based Z3 algorithm on some benchmark formulas that were extracted by the SAGE [68] DSE system. These results suggest that stochastic local search is up to the task of tackling typical program execution path constraints, and justifies its use in SBF to achieve directionality.

In the traditional DSE setting, a local-search based solver would try to find a satisfying solution for a formula by searching the input space with the guidance of a distance function, which is evaluated on the formula. SBF carries out a similar search too, but instead of evaluating the distance of an input assignment on an extracted formula, the distance is computed by executing the instrumented program. Moreover, while the constraint solver would try to satisfy the entire path constraint, we focus on just the last condition. This means that our approach is incremental in generating new inputs: it reuses the part of the existing input that is not influencing the targeted edge and mutates only the part that does. Although these mutations can sometimes cause the program to take an unintended path, our results suggest that the trade-off we make is very beneficial.

The main advantage of SBF, shared with fuzzers, is that it is not fixated on taking a particular path. In particular, if a new mutant that was intended to cover a certain edge ends up going down an unintended path, SBF does not discard it. Instead, if the mutant uncovers new behavior, then it is put back on the work list, and serves as a basis to further increase coverage. In contrast, an external constraint solver cannot make any use of solutions that may very well be useful for discovering new behavior, but do not satisfy the current constraint given to it.

Fuzzing+DSE hybrids. Hybrid approaches [104, 128, 155] can combine some of the benefits of fuzzing and DSE. In these systems, cheap and lightweight fuzzing is used to generate test cases until it gets stuck and does not find new paths any more. DSE is used at this point to "punch through" difficult conditions that the fuzzer was unable to get through. Instead of relying on such an ad-hoc approach for combining the benefits of fuzzing and DSE, SBF uses a more systematic approach by building directionality into fuzzing. Our results demonstrate that SBF's directionality helps it increase coverage at a much faster rate than existing systems.

Note, that SBF can also be used as component of a hybrid system in addition to blind fuzzers and DSE. Adding SBF to an existing Fuzzer+DSE hybrid will make the system as a whole more efficient, as more coverage targets can be reached with cheaper input generation. Namely, most "hard" paths for which symbolic execution was previously necessary, can be quickly covered by SBF.

Search-based software testing (SBST). The idea of using local search algorithms for generating test data has been investigated for a long time [113, 110, 7, 109]. SBST has primarily been used for unit test generation. These tools typically establish the set of paths or branches that they want to reach statically and then target them individually. AUSTIN [92, 93] is one of the most advanced SBST based unit testing tool for C. It targets edges in functions using local search, but uses different scoring function and different search algorithm than SBF. For each edge in the function's control flow graph a new search is initiated. Searches are independent from each other, so the information we gained by generating an input that reaches one edge is lost or unused for generating input to reach another edge along the same path. In contrast, SBF targets edges incrementally, similarly to SAGE's generational search. Also, AUSTIN's search is intra-procedural, meaning that only the nodes inside the function are considered for evaluating the score, while SBF does not have such limitation.

Another example of a real-world tool using SBST is EvoSuite [62]. It is a unit test case generator for Java, using evolutionary algorithm. EvoSuite mutates and scores the entire test suite in one iteration, instead of trying to reach specific coverage targets with individual test cases. The test suite is scored based on the total number of covered edges. In each iteration step, the genetic algorithm treats the entire test suite as the population, and mutates it towards a better (population) score, instead of improving individual test cases.

Vuzzer [134] is a recent fuzzer that also uses evolutionary algorithm to prioritize its input mutation and uses coverage as a scoring function. It also assigns weights to basic blocks using a prior static analysis, and the score (fitness) is calculated using the weights of the covered blocks. This way it operates as a coverage-guided fuzzer but with more detailed coverage information feedback. SBF differs from these tools by targeting potentially reachable new edges using a local search.

Chapter 6

Conclusion

Memory corruption bugs in C/C++ programs continue to be one of the most serious problems in computer security. In this dissertation, we made contributions in both of the two main approaches used to mitigate this problem: in software hardening and in bug finding.

On the hardening side, we introduced a new low-level security policy, called *code-pointer integrity*, and described an effective and efficient mechanism to enforce it. CPI protects systems against all control-flow hijacks that exploit memory bugs. The key idea is to selectively provide full memory safety for just a subset of a program's pointers, namely code pointers. We also introduced CPS, a relaxed form of CPI that still provides stronger guarantees than existing control-flow integrity techniques, for less overhead. Finally, SafeStack can be used to enforce *code-pointer integrity* on the stack, to provide stronger protection than stack cookies, without any overhead. SafeStack is part of the LLVM/Clang compiler and will hopefully become the new default stack protection mechanism.

On the bug-finding side, we introduced *search-based fuzzing*, a new test generation technique that incorporates some of the directionality of symbolic execution with the simplicity and scalability of fuzzers. The key idea is to imitate the operation of a dynamic symbolic execution system with an incomplete constraint solver, but without executing symbolically. We instrument the target program to compute and provide the feedback necessary for a local search based solver, and link such solver with the tested program. We showed that SBF, the tool that implements our target identification, local search based mutation, and test suite inflation&deflation algorithms can be more effective and efficient than similar test generation tools. We make SBF available as an open-source tool that bridges the gap between directed and scalable test generators.

Hardening and bug-finding techniques are equally important and necessary. Program transformations, like CPI, can be used to provide strong protection against exploits with low overhead, which allows applying them for production code. There will always be cases, however, when due to compatibility or performance reasons, such hardening techniques cannot be used. In these cases we can only rely on automated bug-finding tools, such as SBF, to find, and fix, the underlying bugs.

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13:4:1–4:40, November 2009.

[3] Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya, and Meredith Whittaker. Announcing OSS-Fuzz: Continuous Fuzzing for Open Source Software. Google Testing Blog, December 2016.

[4] Periklis Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, pages 12–12, Berkeley, CA, USA, 2010. USENIX Association.

[5] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing Memory Error Exploits with WIT. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 263–277, Washington, DC, USA, 2008. IEEE Computer Society.

[6] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 51–66, Berkeley, CA, USA, 2009. USENIX Association.

[7] S. Ali, L.C. Briand, H. Hemmati, and R.K. Panesar-Walawege. A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation. *Software Engineering, IEEE Transactions on*, 36(6):742–762, Nov 2010.

[8] A. Arcuri, M. Z. Iqbal, and L. Briand. Random Testing: Theoretical Results and Practical Implications. *IEEE Transactions on Software Engineering*, 38(2):258–277, March 2012.

[9] Andrea Arcuri and Gordon Fraser. *On the Effectiveness of Whole Test Suite Generation*, chapter Search-Based Software Engineering: 6th International Symposium, SSBSE 2014, Fortaleza, Brazil, August 26-29, 2014. Proceedings, pages 1–15. Springer International Publishing, Cham, 2014.

[10] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. Formal Analysis of the Effectiveness and Predictability of Random Testing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 219–230, New York, NY, USA, 2010. ACM.

[11] Thanassis Avgerinos, David Brumley, John Davis, Ryan Goulden, Tyler Nighswander, and Alex Rebert. Unleashing the Mayhem CRS. https://blog.forallsecure.com/2016/02/09/unleashing-mayhem/, 2016.

[12] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. Automatic Exploit Generation. *Commun. ACM*, 57(2):74–84, February 2014.

[13] Arthur Baars, Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, Paolo Tonella, and Tanja Vos. Symbolic Search-based Testing. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 53–62, Washington, DC, USA, 2011. IEEE Computer Society.

[14] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A Survey of Symbolic Execution Techniques. *CoRR*, abs/1610.00502, 2016.

[15] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011.

[16] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier. A Taint Based Approach for Smart Fuzzing. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 818–825, April 2012.

[17] Emery D. Berger and Benjamin G. Zorn. DieHard: probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 158–168, New York, NY, USA, 2006. ACM.

[18] Sandeep Bhatkar and R. Sekar. Data Space Randomization. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 1–22, Berlin, Heidelberg, 2008. Springer-Verlag.

[19] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 17–17, Berkeley, CA, USA, 2005. USENIX Association.

[20] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Conflict-driven clause learning SAT solvers. *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications*, pages 131–153, 2009.

[21] Dionysus Blazakis. Interpreter exploitation. In *Proceedings of the 4th USENIX conference on Offensive technologies*, WOOT'10, pages 1–9, Berkeley, CA, USA, 2010. USENIX Association.

[22] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 30–40, New York, NY, USA, 2011. ACM.

[23] Hans-Juergen Boehm and Mark Weiser. Garbage Collection in an Uncooperative Environment. *Softw. Pract. Exper.*, 18(9):807–820, September 1988.

[24] M. Böhme and S. Paul. A Probabilistic Analysis of the Efficiency of Automated Software Testing. *IEEE Transactions on Software Engineering*, 42(4):345–360, April 2016.

[25] Marcel Böhme and Soumya Paul. On the Efficiency of Automated Testing. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 632–642, New York, NY, USA, 2014. ACM.

[26] Marcel Bohme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 23nd ACM SIGSAC Conference on Computer and Communications Security, 2016*, 2016.

[27] Mateus Borges, Marcelo d'Amorim, Saswat Anand, David Bushnell, and Corina S. Pasareanu. Symbolic Execution with Interval Solving and Meta-heuristic Search. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST '12, pages 111–120, Washington, DC, USA, 2012. IEEE Computer Society.

[28] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 114–129, Washington, DC, USA, 2014. IEEE Computer Society.

[29] Derek Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation.* Ph.d. thesis, Massachusetts Institute of Technology, Cambridge, MA, Sep 2004.

[30] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Theory and Techniques for Automatic Generation of Vulnerability-Based Signatures. *IEEE Trans. Dependable Secur. Comput.*, 5(4):224–241, October 2008.

[31] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[32] Cristian Cadar and Koushik Sen. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM*, 56(2):82–90, February 2013.

[33] Nicholas Carlini and David Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, San Diego, CA, August 2014. USENIX Association.

[34] 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*, OSDI '06, pages 147–160, Berkeley, CA, USA, 2006. USENIX Association.

[35] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 45–58, New York, NY, USA, 2009. ACM.

[36] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing Mayhem on Binary Code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 380–394, Washington, DC, USA, 2012. IEEE Computer Society.

[37] Sang Kil Cha, M. Woo, and D. Brumley. Program-Adaptive Mutational Fuzzing. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 725–741, May 2015.

[38] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 559–572, New York, NY, USA, 2010. ACM.

[39] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 12–12, Berkeley, CA, USA, 2005. USENIX Association.

[40] Tzi-cker Chiueh and Fu-Hau Hsu. RAD: A Compile-Time Solution to Buffer Overflow Attacks. In *Distributed Computing Systems, 2001. 21st International Conference on.*, pages 409–417, Los Alamitos, CA, USA, 2001. IEEE, IEEE Computer Society.

[41] Codenomicon. Heartbleed. http://http://heartbleed.com/, 2014.

[42] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: securing software by blocking bad input. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 117–130, New York, NY, USA, 2007. ACM.

[43] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, pages 7–7, Berkeley, CA, USA, 2003. USENIX Association.

[44] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th conference on USENIX Security Symposium - Volume 7*, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.

[45] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. The Performance Cost of Shadow Stacks and Stack Canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, pages 555–566, New York, NY, USA, 2015. ACM.

[46] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416, San Diego, CA, August 2014. USENIX Association.

[47] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: a detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 40–51, New York, NY, USA, 2011. ACM.

[48] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[49] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: architectural support for spatial safety of the C programming language. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 103–114, New York, NY, USA, 2008. ACM.

[50] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 162–171, New York, NY, USA, 2006. ACM.

[51] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. SAFECode: enforcing alias analysis for weakly typed languages. *SIGPLAN Not.*, 41(6):144–157, June 2006.

[52] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory safety without runtime checks or garbage collection. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, LCTES '03, pages 69–80, New York, NY, USA, 2003. ACM.

[53] Artem Dinaburg. Making a scalable automated hacking system. Presented at Infiltrate conference, April 2016.

[54] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. LAVA: Large-scale Automated Vulnerability Addition. In *In IEEE Symposium on Security and Privacy (S&P)*, 2016.

[55] Joshua Drake. Stagefright: Scary Code in the Heart of Android. In *BlackHat USA*, 2015.

[56] Will Drewry and Tavis Ormandy. Flayer: Exposing Application Internals. In *First USENIX Workshop on Offensive Technologies, WOOT '07, Boston, MA, USA, August 6, 2007*, 2007.

[57] Joe W. Duran and S.C. Ntafos. An Evaluation of Random Testing. *Software Engineering, IEEE Transactions on*, SE-10(4):438–444, July 1984.

[58] Tyler Durden. Bypassing PaX ASLR protection. *Phrack*, 11(59), Jul 2002.

[59] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: software guards for system address spaces. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 6–6, Berkeley, CA, USA, 2006. USENIX Association.

[60] Hiroaki Etoh and Kunikazu Yoda. Protecting from stack-smashing attacks. Published on World-Wide Web at URL http://www.trl.ibm.com/projects/security/ssp/main.html, 2000.

[61] Roger Ferguson and Bogdan Korel. The Chaining Approach for Software Test Data Generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, January 1996.

[62] Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic Test Suite Generation for Object-oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. ACM.

[63] Andreas Frohlich, Armin Biere, Christoph M. Wintersteiger, and Youssef Hamadi. Stochastic Local Search for Satisfiability Modulo Theories. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2015)*. AAAI - Association for the Advancement of Artificial Intelligence, January 2015.

[64] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based Directed Whitebox Fuzzing. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 474–484, Washington, DC, USA, 2009. IEEE Computer Society.

[65] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL (T): Fast decision procedures. In *International Conference on Computer Aided Verification*, pages 175–188. Springer, 2004.

[66] Patrice Godefroid. Micro Execution. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 539–549, New York, NY, USA, 2014. ACM.

[67] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated Random Testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.

[68] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*, 2008.

[69] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of Control: Overcoming Control-Flow Integrity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 575–589, Washington, DC, USA, 2014. IEEE Computer Society.

[70] Peter Goodman. A fuzzer and a symbolic executor walk into a cloud. https://blog.trailofbits.com/2016/08/02/engineering-solutions-to-hard-program-analysis-problems, 2016.

[71] Jordan Gruskovnjak and VUPEN Vulnerability Research Team. Advanced Exploitation of Mozilla Firefox Use-after-free Vulnerability (MFSA 2012-22), June 2012.

[72] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 49–64, Washington, D.C., 2013. USENIX.

[73] N. Hasabnis, A. Misra, and R. Sekar. Light-weight Bounds Checking. 2012.

[74] John L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[75] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. ILR: Where'd My Gadgets Go? In *IEEE Symposium on Security and Privacy*, pages 571–585, 2012.

[76] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with Code Fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 38–38, Berkeley, CA, USA, 2012. USENIX Association.

[77] Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. Profile-guided Automated Software Diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society.

[78] Holger H Hoos and Thomas Stützle. *Stochastic local search: Foundations & applications*. Elsevier, 2004.

[79] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986, May 2016.

[80] huku and argp. Exploiting VLC: A case study on jemalloc heap overflows. *Phrack*, 2012.

[81] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks Against Kernel Space ASLR. In *IEEE Symposium on Security and Privacy*, 2013.

[82] Intel. Introduction to Intel Memory Protection Extensions. https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions, July 2013.

[83] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, ATEC '02, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[84] R.W.M. Jones and P.H.J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. *Automated and Algorithmic Debugging*, 25, 1997.

[85] jp. Advanced Doug Lea's malloc exploits. *Phrack*, 11(61), Aug 2003.

[86] Rauli Kaksonen. *A functional method for assessing protocol implementation security.* Technical Research Centre of Finland, 2001.

[87] 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*, CCS '03, pages 272–280, New York, NY, USA, 2003. ACM.

[88] Vasileios P. Kemerlis. kGuard: Lightweight Kernel Protection against Return-to-User Attacks, 2012.

[89] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, ACSAC '06, pages 339–348, Washington, DC, USA, 2006. IEEE Computer Society.

[90] B. Korel. Automated Software Test Data Generation. *IEEE Trans. Softw. Eng.*, 16(8):870–879, August 1990.

[91] Bogdan Korel. Dynamic method for software test data generation. *Software Testing, Verification and Reliability*, 2(4):203–213, 1992.

[92] Kiran Lakhotia. *Search-Based Testing.* PhD thesis, King's College London, 2009.

[93] Kiran Lakhotia, Mark Harman, and Hamilton Gross. AUSTIN: An Open Source Tool for Search Based Software Testing of C Programs. *Inf. Softw. Technol.*, 55(1):112–125, January 2013.

[94] Kiran Lakhotia, Nikolai Tillmann, Mark Harman, and Jonathan De Halleux. FloPSy: Search-based Floating Point Constraint Solving for Symbolic Execution. In *Proceedings of the 22Nd IFIP WG 6.1 International Conference on Testing Software and Systems*, ICTSS'10, pages 142–157, Berlin, Heidelberg, 2010. Springer-Verlag.

[95] Andrea Lanzi, Lorenzo Martignoni, Mattia Monga, and Roberto Paleari. A Smart Fuzzer for x86 Executables. In *Proceedings of the Third International Workshop on Software Engineering for Secure Systems*, SESS '07, pages 7–, Washington, DC, USA, 2007. IEEE Computer Society.

[96] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. SoK: Automated Software Diversity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 276–291, Washington, DC, USA, 2014. IEEE Computer Society.

[97] Chris Lattner. *Macroscopic Data Structure Analysis and Optimization.* PhD thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, May 2005. *See* `http://llvm.cs.uiuc.edu`.

[98] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[99] Chris Lattner and Vikram Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 129–142, New York, NY, USA, 2005. ACM.

[100] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making Context-sensitive Points-to Analysis with Heap Cloning Practical for the Real World. *SIGPLAN Not.*, 42(6):278–289, June 2007.

[101] Jinku Li, Zhi Wang, Tyler K. Bletsch, Deepa Srinivasan, Michael C. Grace, and Xuxian Jiang. Comprehensive and Efficient Protection of Kernel Control Data. *IEEE Transactions on Information Forensics and Security*, 6(4):1404–1417, dec 2011.

[102] Guangcheng Liang, Lejian Liao, Xin Xu, Jianguang Du, Guoqiang Li, and Henglong Zhao. Effective Fuzzing Based on Dynamic Taint Analysis. In *Computational Intelligence and Security (CIS), 2013 9th International Conference on*, pages 615–619, Dec 2013.

[103] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[104] Rupak Majumdar and Koushik Sen. Hybrid Concolic Testing. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 416–426, Washington, DC, USA, 2007. IEEE Computer Society.

[105] Felipe Andres Manzano. The Symbolic Maze! Feliam's Blog, October 2010.

[106] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing System Virtual Machines. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 171–182, New York, NY, USA, 2010. ACM.

[107] Ali José Mashtizadeh, Andrea Bittau, David Mazières, and Dan Boneh. Cryptographically Enforced Control Flow Integrity. *CoRR*, abs/1408.1451, 2014.

[108] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC architecture. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[109] P. McMinn. Search-Based Software Testing: Past, Present and Future. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 153–163, March 2011.

[110] Phil McMinn. Search-based Software Test Data Generation: A Survey: Research Articles. *Softw. Test. Verif. Reliab.*, 14(2):105–156, June 2004.

[111] Microsoft. The BlueHat Prize. https://www.microsoft.com/security/bluehatprize, 2012.

[112] Barton P. Miller, Louis Fredriksen, and Bryan So. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM*, 33(12):32–44, December 1990.

[113] W. Miller and D. L. Spooner. Automatic Generation of Floating-Point Test Data. *IEEE Transactions on Software Engineering*, SE-2(3):223–226, Sept 1976.

[114] MITRE. CWE/SANS Top 25 Most Dangerous Software Errors, Sept. 2011.

[115] Santosh Nagarakatte and Milo MK Martin Steve Zdancewic. WatchdogLite: Hardware-Accelerated Compiler-Based Pointer Checking. 2014.

[116] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: highly compatible and complete spatial memory safety for c. *SIGPLAN Not.*, 44(6):245–258, June 2009.

[117] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: compiler enforced temporal safety for C. In *Proceedings of the 2010 international symposium on Memory management*, ISMM '10, pages 31–40, New York, NY, USA, 2010. ACM.

[118] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, pages 128–139, New York, NY, USA, 2002. ACM.

[119] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack*, 11(58), Dec 2001.

[120] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[121] Ducson Nguyen. Hybrid Concolic Execution. http://blogs.grammatech.com/hybrid-concolic-execution-part-1, 2013.

[122] Ben Niu and Gang Tan. Monitor Integrity Protection with Space Efficiency and Separate Compilation. In *20th ACM Conference on Computer and Communication Security (CCS '13)*, 2013.

[123] Ben Niu and Gang Tan. Modular Control-flow Integrity. *SIGPLAN Not.*, 49(6):577–587, June 2014.

[124] Gene Novark and Emery D. Berger. DieHarder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 573–584, New York, NY, USA, 2010. ACM.

[125] P. Oehlert. Violating assumptions with fuzzing. *IEEE Security Privacy*, 3(2):58–62, March 2005.

[126] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-Free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 49–58, New York, NY, USA, 2010. ACM.

[127] M. Oriol. Random Testing: Evaluation of a Law Describing the Number of Faults Found. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 201–210, April 2012.

[128] Brian S. Pak. Hybrid Fuzz Testing: Discovering Software Bugs via Fuzzing and Symbolic Execution. Master's thesis, Carnegie Mellon University, 2012.

[129] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 601–615, Washington, DC, USA, 2012. IEEE Computer Society.

[130] Team PaX. Address Space Layout Randomization, July 2001.

[131] Mathias Payer. Too much PIE is bad for performance, 2012.

[132] Phoronix. Phoronix Test Suite. http://www.phoronix-test-suite.com/, 2008.

[133] Rui Qiao, Mingwei Zhang, and R. Sekar. A Principled Approach for ROP Defense. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, pages 101–110, New York, NY, USA, 2015. ACM.

[134] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *NDSS*, November 2017.

[135] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing Seed Selection for Fuzzing. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 861–875, San Diego, CA, August 2014. USENIX Association.

[136] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Trans. Info. & System Security*, 15(1), March 2012.

[137] Marc Roper, Iain Maclean, Andrew Brooks, James Miller, and Murray Wood. Genetic algorithms and the automatic generation of test data. *University of Strathclyde, UK*, 1995.

[138] Olatunji Ruwase and Monica S. Lam. A Practical Dynamic Buffer Overflow Detector. In *In Proceedings of the 11th Annual Network and Distributed System Security Symposium*, pages 159–169, 2004.

[139] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic Superoptimization. *SIGPLAN Not.*, 48(4):305–316, March 2013.

[140] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, February 2000.

[141] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 317–331, Washington, DC, USA, 2010. IEEE Computer Society.

[142] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit Hardening Made Easy. In *Proceedings of the USENIX Security Symposium*, 2011.

[143] Bart Selman, Henry A. Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In *Cliques, Coloring, and Satisfiability, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, October 11-13, 1993*, pages 521–532, 1993.

[144] Bart Selman, Hector Levesque, and David Mitchell. A New Method for Solving Hard Satisfiability Problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, AAAI'92, pages 440–446. AAAI Press, 1992.

[145] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[146] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC 2012*, 2012.

[147] Kostya Serebryany. Simple guided fuzzing for libraries using LLVM's new libFuzzer. http://blog.llvm.org/2015/04/fuzz-all-clangs.html, 2015.

[148] Fermin J. Serna. CVE-2012-0769, the case of the perfect info leak, 2012.

[149] 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*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[150] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[151] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[152] Stelios Sidiroglou-Douskos, Eric Lahtinen, Nathan Rittenhouse, Paolo Piselli, Fan Long, Deokhwan Kim, and Martin Rinard. Targeted Automatic Integer Overflow Discovery Using Goal-Directed Conditional Branch Enforcement. *SIGPLAN Not.*, 50(4):473–486, March 2015.

[153] Asia Slowinska, Traian Stancescu, and Herbert Bos. Body armor for binaries: preventing buffer overflows without recompilation. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 11–11, Berkeley, CA, USA, 2012. USENIX Association.

[154] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 574–588, Washington, DC, USA, 2013. IEEE Computer Society.

[155] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *NDSS*, 2016.

[156] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*, EUROSEC '09, pages 1–8, New York, NY, USA, 2009. ACM.

[157] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[158] Ari Takanen, Jared D Demott, and Charles Miller. *Fuzzing for software security testing and quality assurance*. Artech House, 2008.

[159] Adobe Product Security Incident Response Team. Fuzzing Reader. https://blogs.adobe.com/security/2009/12/fuzzing_reader_-_lessons_learned.html, 2009.

[160] Google Chrome Team. Fuzzing for Security. https://blog.chromium.org/2012/04/fuzzing-for-security.html, 2012.

[161] Microsoft SDL Team. Fuzz Testing at Microsoft and the Triage Process. https://blogs.microsoft.com/microsoftsecure/2007/09/20/fuzz-testing-at-microsoft-and-the-triage-process/, 2007.

[162] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955, San Diego, CA, August 2014. USENIX Association.

[163] Nigel Tracey, John Clark, and Keith Mander. Automated Program Flaw Finding Using Simulated Annealing. *SIGSOFT Softw. Eng. Notes*, 23(2):73–81, March 1998.

[164] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the expressiveness of return-into-libc attacks. In *Proceedings of the 14th international conference on Recent Advances in Intrusion Detection*, RAID'11, pages 121–141, Berlin, Heidelberg, 2011. Springer-Verlag.

[165] P. Tsankov, M. T. Dashti, and D. Basin. SECFUZZ: Fuzz-testing security protocols. In *Automation of Software Test (AST), 2012 7th International Workshop on*, pages 1–7, June 2012.

[166] Arjan Van de Ven and Ingo Molnar. Exec Shield, 2004.

[167] Vendicator. Stack Shield: A "stack smashing" technique protection tool for Linux, 2000.

[168] Dmitry Vyukov. syzkaller - linux syscall fuzzer. `https://github.com/google/syzkaller`, 2014.

[169] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, SOSP '93, pages 203–216, New York, NY, USA, 1993. ACM.

[170] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 497–512, May 2010.

[171] Richard Wartell, Vishwath Mohan, Kevin Hamlen, and Zhiqiang Lin. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS'12)*, Raleigh, NC, October 2012.

[172] Joachim Wegener, André Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information & Software Technology*, 43(14):841–854, 2001.

[173] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. RIPE: runtime intrusion prevention evaluator. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 41–50, New York, NY, USA, 2011. ACM.

[174] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling Black-box Mutational Fuzzing. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pages 511–522, New York, NY, USA, 2013. ACM.

[175] Wei Xu, Daniel C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, SIGSOFT '04/FSE-12, pages 117–126, New York, NY, USA, 2004. ACM.

[176] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011.

[177] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: a sandbox for portable, untrusted x86 native code. *Commun. ACM*, 53(1):91–99, January 2010.

[178] Suan Hsi Yong and Susan Horwitz. Protecting C programs from attacks via invalid pointer dereferences. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-11, pages 307–316, New York, NY, USA, 2003. ACM.

[179] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. PAriCheck: an efficient pointer arithmetic checker for C programs. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 145–156, New York, NY, USA, 2010. ACM.

[180] Michal Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl, 2014.

[181] Michal Zalewski. Bash bug: the other two RCEs, or how we chipped away at the original fix. https://lcamtuf.blogspot.com/2014/10/bash-bug-how-we-finally-cracked.html, 2014.

[182] Michal Zalewski. Pulling JPEGs out of thin air. https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html, 2014.

[183] Bin Zeng, Gang Tan, and Úlfar Erlingsson. Strato: A Retargetable Framework for Low-Level Inlined-Reference Monitors. In *USENIX Security Symposium*, 2013.

[184] Bin Zeng, Gang Tan, and Greg Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 29–40, New York, NY, USA, 2011. ACM.

[185] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical Control Flow Integrity & Randomization for Binary Executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 559–573, May 2013.

[186] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R. Sekar. A Platform for Secure Static Binary Instrumentation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pages 129–140, New York, NY, USA, 2014. ACM.

[187] Mingwei Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 337–352, Berkeley, CA, USA, 2013. USENIX Association.

[188] Mingwei Zhang and R. Sekar. Control Flow and Code Integrity for COTS Binaries: An Effective Defense Against Real-World ROP Attacks. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, pages 91–100, New York, NY, USA, 2015. ACM.

[189] Mingyi Zhao and Peng Liu. Empirical Analysis and Modeling of Black-Box Mutational Fuzzing. In *Engineering Secure Software and Systems: 8th International Symposium, ESSoS 2016, London, UK, April 6-8, 2016. Proceedings*, pages 173–189. Springer International Publishing, Cham, 2016.