KTH ROYAL INSTITUTE
OF TECHNOLOGY

Doctoral Thesis in Information and Communication Technology

Generating Optimized and Secure Binary Code

RODOTHEA MYRSINI TSOUPIDI

Stockholm, Sweden 2023

Generating Optimized and Secure Binary Code

RODOTHEA MYRSINI TSOUPIDI

Academic Dissertation which, with due permission of the KTH Royal Institute of Technology, is submitted for public defence for the Degree of Doctor of Philosophy on Wednesday the 7th June 2023, at 1:00 p.m. in F3, Lindstedtsvägen 26, Stockholm.

Doctoral Thesis in Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden 2023

ISBN: 978-91-8040-591-1
TRITA-EECS-AVL-2023:44

Printed by: Universitetsservice US-AB, Sweden 2023

Abstract

The increased digitalization of modern societies has resulted in a proliferation of a broad spectrum of embedded devices, ranging from personal smartphones and heart pacemakers to large-scale industrial IoT systems. Since they often handle various sensitive data, these devices increasingly become the targets of cyberattacks that threaten the integrity of personal data, financial security, and sometimes even people's safety.

A common source of security vulnerabilities in computing systems is software. Nowadays, the vast majority of embedded software is written in high-level programming languages and compiled to low-level assembly code using general-purpose compilers. However, general-purpose compilers typically ignore security aspects and mainly focus on improving performance and reducing the code size. Meanwhile, the security-targeting compilers often produce code that is suboptimal with respect to performance. This security-performance gap is particularly detrimental for embedded devices that are usually battery-operated and hence, have stringent restrictions on memory size and power consumption.

Among the most frequently carried out cyberattacks are code-reuse attacks. They insert data into the victim system via memory-corruption vulnerabilities to redirect the control flow and hijack the system. Automatic software diversification is an efficient mitigation approach against code-reuse attacks, however, it typically does not allow us to explicitly control of the introduced performance overhead.

Another large class of attacks is side-channel attacks. Such attacks often target cryptographic implementations and aim at extracting the information about the processed data by recording side-channel information, such as the execution time or the power consumption of the victim system. Typically, protection against side-channel attacks relies on software-based mitigations, which may lead to high performance overhead. An attacker that attempts to hijack the victim system may use either or both of these attacks and hence, often multiple mitigations have to be combined together to protect a system.

This dissertation proposes Secure-by-Construction Optimization (SecOpt), a constraint-based approach that combines performance goals with security mitigations. More specifically, SecOpt achieves performance-aware automatic code diversification against code-reuse attacks, while it generates highly-optimized code that preserves software mitigations against side-channel attacks. A key advantage of SecOpt is composability, namely the ability to combine conflicting mitigations and generate code that preserves these mitigations. In particular, SecOpt generates diverse code variants that are secure against side-channel attacks, therefore protecting against both code-reuse and side-channel attacks.

SecOpt features unique characteristics compared to conventional compiler-based approaches, including performance-awareness and mitigation composability in a formal framework. Since the combined security and performance goals are especially important for resource-constrained systems, SecOpt constitutes a practical approach for optimizing performance- and security-critical code for embedded devices.

Sammanfattning

Den ökande digitaliseringen av det moderna samhället har orsakat snabb spridning av ett brett utbud av inbyggda system, allt från smarta mobiltelefoner och hjärtstimulatorer, till storskaliga industriella IoT system. Dessa datorenheter blir allt oftare mål för cyberangrepp som hotar den personliga integriteten, den ekonomiska säkerheten och ibland även människors säkerhet.

En vanlig källa till säkerhetssårbarheter i datasystem är mjukvara. Nu för tiden är majoriteten av mjukvaran för inbyggda system skriven i högnivåprogrammeringsspråk som kompileras till maskinkod med hjälp av konventionella kompilatorer. Dessa kompilatorer tar ofta inte hänsyn till säkerhetsaspekter i programmets källkod och fokuserar istället på att förbättra prestanda och reducera kodstorlek. Samtidigt producerar säkerhetsinriktade kompilatorer ofta kod som är suboptimal med avseende på prestanda. Denna diskrepans mellan säkerhet och prestanda är problematisk för inbyggda system med stränga restriktioner vad gäller minnesanvändning och energiförbrukning.

Kodåteranvändningsattacker är en av de vanligaste typer av cyberangrepp. Dessa cyberangrepp injicerar data i det angripna systemet, via en minneskorruptionssårbarhet, som ger möjlighet att dirigera om mjukvarans kontrollflöde och kapa systemet. Automatiserad mjukvarudiversifiering är en effektiv skyddsåtgärd mot kodåteranvändningsattacker men nuvarande metoder tillåter inte explicit styrning av prestandaförsämringen. En annan stor cyberangreppsklass är sidokanalsattacker. Dessa cyberangrepp riktas ofta mot kryptografiska implementeringar och syftar till att utvinna säkerhetsviktig information som berör den behandlade datan. Angriparen läser av sidokanalsinformation under programmets exekvering, såsom exekveringstid eller energiförbrukning. Vanliga skyddsåtgärder mot sidokanalsattacker är mjukvaruåtgärder, som dessvärre kan leda till stor prestandaförsämring. En angripare som försöker kapa ett system kan använda en eller flera metoder för att utföra dessa cyberangrepp. Därför måste ofta olika skyddsåtgärder kombineras för att skydda ett system.

Denna avhandling introducerar Säker-vid-Konstruktion Kodoptimering (SecOpt), en villkorsbaserad kompileringsmetod som kombinerar prestandamål med skyddsåtgärder. Närmare bestämt utför SecOpt prestandamedveten automatisk diversifiering mot kodåteranvändningsattacker och genererar optimerad kod som bibehåller mjukvaruåtgärder mot sidokanalsattacker. SecOpts nykelegenskap är dess möjlighet att kombinera motstridiga skyddsåtgärder på ett sätt som bevarar dessa skyddsåtgärders egenskaper. Mer specifikt skapar SecOpt mångfaldiga kodvarianter som uppfyller säkerhetskrav mot sidokanalsattacker, vilket skyddar både mot kodåteranvändningsattacker och sidokanalsattacker.

SecOpt har unika egenskaper jämfört med konventionella kompileringsmetoder, såsom prestandamedvetenhet och komponering av olika skyddsåtgärder i ett formellt ramverk. Kombinationen av säkerhets- och prestandamål är särskilt viktig för resursbegränsade inbyggda system. Sammanfatningsvis är SecOpt en praktisk metod för att optimera säkerhetskritisk kod.

To my grandparents Myrsini and Stratis

Acknowledgments

First, I would like to thank my main supervisor Elena Troubitsyna, who trusted my work and supported me during my PhD. This dissertation would not have been possible without her trust, support, research advice, and supervision. I would also like to thank my co-advisor Panos Papadimitratos, who helped me focus on the security angle of my work and the presentation of my research results. I want to thank my former co-advisor Roberto Castañeda Lozano for teaching me a lot about presenting and writing research, supporting me during my studies, and continuing to advise and work with me until the end of my studies. A special thanks to Christian Schulte, who taught me all I know about Constraint Programming and gave me the opportunity to work on an exciting project that is the basis of this dissertation. You were an inspiration for my work, and you are dearly missed. In addition, I am especially grateful to Thomas Sjöland for his significant and continuous support. I want to thank Fernando Magno Quintão Pereira for serving as the opponent of this dissertation, Elisavet Kozyri, Christoph Kessler, and Marjan Sirjani for serving on the grading committee, Vladimir Vlassov for serving as a chair at my defense, and Roberto Guanciale for acting as the advanced reviewer for my thesis.

Big thanks to my friends and colleagues Amir M. Ahmadian, Nicolas Harrand, and Javier Cabrera Arteaga for improving the quality of my work and the quality of my life with many discussions, fikas, and activities inside and outsite KTH. I also want to thank my friends and colleagues, Saranya Natarajan, Alexandros Milolidakis, Orestis Floros, Nadia Campo Woytuk, Negar Sarinianaini, Andreas Lindner, Deepika Tiwari, Anoud Alshnakat, Daniel Lundén, Gizem Çaylak, Tianze Wang, Han Fu, Linnea Stjerna, Javier Ron Arteaga, and Viktor Palmkvist, for the lunch and fika conversations, board-game nights, and taking care of my cats! I want to especially thank my parents, my grandmother Myrsini and my grandfather Stratis, my sister Sofia, my two brothers Panagiotis and Stratis, and recently my niece Marielsa, for their endless support and trust in me. Last but not least, I want to thank Oscar for all his continuous support, patience, and for sharing good and bad moments.

Contents

Thesis

Chapter 1

Introduction

The increasing digitalization of modern societies has created the need to protect sensitive data and safety-critical systems from malicious actors. Embedded devices, such as medical implants, traffic controllers, and car microcontrollers, are safety-critical systems that require preserving strict safety and security requirements. In addition, embedded devices are often battery driven, and thus, resource constrained [1].

Despite long-term efforts to identify and remove security vulnerabilities in software systems, computer systems are still vulnerable to security threats. These vulnerabilities are the result of design decisions, human errors, connectivity via public network, public exposure of the hardware, and/or insufficient control to ensure information security [2]. Furthermore, embedded software is typically written in unsafe languages, such as C and C++, which supports a wide range of target architectures. Mitigating these security vulnerabilities requires vulnerable software and/or hardware changes. Changes in the software are easier to implement and deliver than hardware due to the long development process of hardware. Software security mitigation approaches often apply changes to the source code of software implementations, which is typically compiled to machine code in binary format.

General-purpose compilers focus on the generated codes' performance efficiency size; however, they typically do not preserve security properties [3]. In recent years, secure compilers have aimed to fill this gap and automatically generate secure code. Unfortunately, these approaches often introduce significant overhead to the performance or code size of the generated code [4]. Resource-constrained Internet of Things (IoT) devices and embedded systems may not afford software mitigations that introduce high performance and code-size overhead. This performance-security gap in compiler approaches creates the need for approaches that mitigate cyber-attacks while generating highly optimized code. Constraint-based modeling and solving is a naturally versatile framework that allows the expression of diverse properties. Constraint-based compiler methods trade high-optimality guarantees, composability, and formal guarantees for compilation time [5].

Among the most powerful software-induced cyberattacks are code-reuse attacks [6]. These attacks insert data into the victim system via memory-corruption vulnerabilities to redirect the control flow and hijack the system. Automatic software diversification investigates the automatic generation of diverse program variants and is an efficient mitigation approach against code-reuse attacks [7]. Although the reported performance overhead is typically low [8], few approaches allow control over the introduced performance overhead [9] to generate diverse variants with predictable performance overhead. Furthermore, most automatic software diversification approaches do not provide any guarantees on the diversity of the generated program variants.

Cryptographic algorithms aim at securing sensitive information and communication in the presence of adversarial behavior. The design of popular cryptographic algorithms, such as RSA, is based on the assumption that breaking these algorithms is computationally too hard to be practical. However, with the advent of side-channel attacks, adversaries are able to break cryptographic algorithms using knowledge about the algorithm's implementation and side effects during the execution of the algorithm. In particular, these attacks exploit side-channel information, such as the execution time or the power consumption, during the execution of the victim algorithm to extract information about the processed data. These powerful attacks have challenged the security of popular cryptographic algorithms, such as AES, DES, and RSA [10, 11, 12]. Typical mitigations against side-channel attacks include software mitigations that aim at hiding secret information from side-channel traces. However, these mitigations may lead to high performance overhead. Hence, reducing this overhead is essential, not least for resource-constrained devices.

An attacker attempting to exploit a victim system may use either or both of these attacks; therefore, protecting a system often requires applying multiple mitigations. However, these mitigations may affect or invalidate each other; thus, combining such mitigations while preserving their security properties is highly important. At the same time, the sequential application of software mitigations may introduce prohibively high performance overhead; hence, controlling this overhead is essential, especially for resource-constrained devices.

This dissertation investigates the generation of highly optimized and secure binary code targeting code-reuse and side-channel attacks in embedded systems. Compiler approaches allow high control over the program structure and the generated binary code, which enables effective vulnerability mitigation. In addition, typically, compilers generate optimized code; thus, embedding security properties in a compiler-base approach allows the generation of highly optimized code [4]. This dissertation presents Secure-by-Construction Optimization (SecOpt), a performance-aware, secure compilation method, which uses Constraint Programming (CP), a combinatorial optimization method, to generate highly optimized and secure code. Compiler optimization has effectively used CP to describe the program properties, code transformations, and the target processor cost model [5]. SecOpt extends a constraint-based compiler approach to generate code that hinders cyberattacks while it generates highly efficient code at the cost of compilation-time overhead.

Code verification allows verifying security properties in the generated binary code to increase trust in the compiler result. The ultimate goal of SecOpt is to design a versatile compiler-based toolbox that protects binary code against code-reuse and side-channel attacks while reducing the introduced resource overhead.

1.1 Thesis Statement

This dissertation proposes a constraint programming approach to integrate compiler transformations and security constraints to generate optimized and secure code. The thesis statement of this dissertation is the following:

> Combinatorial binary-code hardening is effective, composable, and highly optimizing.

The proposed approach is *effective* as it achieves satisfactory mitigation effectiveness against different attacks, including code-reuse attacks, power side-channel attacks, and timing side-channel attacks. When the attacker has multiple methods to hijack a system, this approach may *compose* one solution that satisfies mitigations against all these threats. This property is valuable when two different mitigation approaches conflict, i.e. the transformations of one mitigation may invalidate the other mitigation(s). SecOpt is *highly optimizing* because it achieves software diversification with zero performance overhead and generates optimized code against side-channel attacks with reduced overhead compared to related approaches. These properties of SecOpt take advantage of the characteristics of CP, which allows control over both the modeling and solving.

1.2 Research Questions

This dissertation poses four research questions that investigate the feasibility and effectiveness of SecOpt at generating highly optimized secure programs, providing a formal and composable framework.

RQ1: How feasible and effective is performance-aware constrained-based software diversification against code-reuse attacks?

Automatic software diversity has been effective against code-reuse attacks. Fine-grained diversification approaches perform transformations at the binary or the compiler level to generate functionally equivalent program variants. However, most of these approaches 1) focus on x86 systems, 2) do not control how different the generated variants are, and/or 3) do not control the performance overhead of the generated program variants. With this question, we want to investigate the feasibility of a constraint-based diversification approach and its effectiveness against

code-reuse attacks. In addition, we investigate how to generate highly optimized and diverse solutions efficiently.

RQ2: How feasible is secure constraint-based optimization of cryptographic implementations?

Software transformations of cryptographic implementations that mitigate timing and power side-channel attacks may introduce significant performance overhead [13, 14, 15]. This dissertation considers two software mitigation approaches against timing and power side channels, respectively and investigates the feasibility of a constraint-based approach to generate optimized and secure code against these attacks. In addition, this question investigates the adequacy of a constraint-based approach to provide guarantees about the program security.

RQ3: How feasible and effective is a combined mitigation against code-reuse attacks and side-channel attacks?

Protecting a system against cyberattacks often requires combining multiple mitigations against different attack classes. In some cases, diverse mitigations may conflict with each other. In particular, the sequential application of different security mitigations may invalidate one another. This dissertation investigates the feasibility of a constraint-based approach to combine multiple mitigations and the security effect of combining fine-grained software diversification against code-reuse attacks with mitigations against side-channel attacks.

RQ4: How feasible is code verification of binary code against timing side channels?

The code that SecOpt generates against timing attacks needs to preserve timing properties. To improve our trust in SecOpt, we verify the intended timing properties in the generated code using external tools. Furthermore, this dissertation investigates the feasibility of a symbolic execution approach for verifying timing mitigations in WebAssembly. WebAssembly is a recent low-level language with multiple advantages, including security features, portability, and efficiency [16]. However, WebAssembly is vulnerable to timing side-channel attacks. This question aims to investigate the feasibility of code verification to preserve timing properties in real-world binary code.

1.3 Contributions

The contributions of this thesis are as follows:

C1: design and evaluate a local search algorithm for generating diverse solutions in CP;

C2: propose a software diversification method that allows explicit control over the execution-time overhead of the generated program variants and evaluate its effectiveness against code-reuse attacks;

C3: design and evaluate a structural decomposition algorithm to diversify medium-sized functions;

C4: model the automatic generation of optimized code that is secure against power side-channel attacks;

C5: provide a proof that the constraint model against power side-channel attacks protects against the leakage model;

C6: model the automatic generation of optimized code that is secure against timing side-channel attacks;

C7: model and evaluate a composable approach to code optimization that is secure against code-reuse attacks and side-channel attacks;

C8: design and evaluate a method to verify the constant-time property in WebAssembly programs.

1.4 Sustainability and Ethics

Sustainability goals and ethics considerations are an essential part of this thesis that aims at extending the state-of-the-art in secure code generation.

Sustainability: Cybersecurity is essential for preventing disinformation, fraud, and breach of sensitive data, while it promotes economic growth and political independence. This thesis concerns the development of software mitigations against cyberattacks that may result in the leakage of sensitive data or hijacking a possibly critical system, such as medical equipment, energy production, medical records, and more. In addition, the energy consumption of data centers accounts for around 1% to 1.5% of global energy use. Often, this data requires security measures to protect against variable attacker models, which typically increase the performance overhead, and thus, the total energy consumption for achieving the same results. We believe that our approach is a step towards improved performance consumption for software and, hence, reduced energy consumption.

Ethics: This thesis deals with data that consists of programs that are not subject to ethical considerations. The reproducibility of the research conducted during this thesis has been an important goal that we deal with by providing all artifacts for this work online.

1.5 Publications

This thesis includes the following publications:

P1: R. M. Tsoupidi, R. Castañeda Lozano, and B. Baudry, "Constraint-Based Software Diversification for Efficient Mitigation of Code-Reuse Attacks," in *International Conference on Principles and Practice of Constraint Programming*, 2020, pp. 791–808

Contributions: The author of this thesis contributed with design discussions, design decisions, implementation and evaluation of the method, paper writing, and presentation of the paper.

P2: R. M. Tsoupidi, R. Castañeda Lozano, and B. Baudry, "Constraint-based diversification of JOP gadgets," *Journal of Artificial Intelligence Research*, vol. 72, pp. 1471–1505, 2021

Contributions: The author of this thesis contributed with design discussions, algorithm design decisions, implementation and evaluation of the method, and paper writing.

P3: R. M. Tsoupidi, M. Balliu, and B. Baudry, "Vivienne: Relational Verification of Cryptographic Implementations in WebAssembly," in *2021 IEEE Secure Development Conference (SecDev)*, 2021, pp. 94–102

Contributions: The author of this thesis contributed with design discussions, algorithm design decisions, solver optimization decisions, invariant design, implementation and evaluation of all methods, paper writing, and presentation of the paper.

P4: R. M. Tsoupidi, R. Castañeda Lozano, E. Troubitsyna, and P. Papadimitratos, "Securing Optimized Code Against Power Side Channels," in *2023 IEEE Security Foundations Symposium (CSF)*, 2023, to appear

Contributions: The author of this thesis contributed with design discussions, algorithm design decisions, search algorithms, proof design and implementation, implementation and evaluation of all methods, paper writing, and future presentation of the paper.

P5: R. M. Tsoupidi, E. Troubitsyna, and P. Papadimitratos, "Thwarting code-reuse and side-channel attacks in embedded systems," *arXiv preprint arXiv:2304.13458*, 2023

Contributions: The author of this thesis contributed with timing side-channel modeling, search algorithms, implementation and evaluation of all methods, paper writing, and potential presentation of the paper.

Table 1.1 shows the research questions for each paper, and Table 1.2 shows the contributions of each paper.

publication	R1	R2	R3	R4
A	✓			
B	✓			
C				✓
D		✓		
E	✓	✓	✓	✓

Table 1.1: Research questions addressed per publication

publication	C1	C2	C3	C4	C5	C6	C7	C8
A	✓	✓						
B		✓	✓					
C								✓
D				✓	✓			
E		✓				✓	✓	

Table 1.2: Contributions per publication

1.6 Outline

Chapter 2 discusses the background of this dissertation, including the cyberattacks that this dissertation considers, the combinatorial approach of this work, and the underlying constraint-based compiler backend. Chapter 3 describes the approach and methodology of this dissertation, while Chapter 4 discusses the related work. Chapter 5 presents the summary of the publications that this dissertation includes, and finally, Chapter 6 concludes this dissertation and discusses potential future-work directions.

Chapter 2

Background

This chapter presents the background of this dissertation. The background includes a description of the cyberattacks and mitigations of these cyberattacks that we consider in this dissertation (Section 2.1), a summary of CP, the primary solving method in this dissertation (Section 2.2), and finally, a description of the modeling of the combinatorial compiler that significant part of this dissertation lies upon (Section 2.3).

2.1 Cybersecurity Threats and Mitigations

Cybersecurity is an increasingly important scientific field due to the emergence of IoT devices and the digitalization of services, including transmission and storage sensitive medical information, safety-critical infrastructure, and financial transactions. Cyberattacks constitute a severe threat in modern societies because these attacks lead to economic loss, negative reputation effects, and negative social impact [22]. Many of these attacks are due to vulnerabilities in software or unintended behavior in the hardware [23].

An important stage in the modern software development chain is compilation. Typically, compilers translate code from a high-level language to a low-level language, such as the binary code that the processor executes. Compilers aim at generating code that is semantically equivalent to the source code, however, there is no requirement to preserve security properties. On the contrary, compilers have in some cases been found responsible for removing security software countermeasures or violating source-code security properties [3]. The reason for these security violations is that compiler research has focused on improving performance and/or code size, whereas security is a concept that has become relevant in recent years due to the increasing use of electronic transactions and IoT devices.

Code-reuse and side-channel attacks are two types of powerful attacks where compilers play an essential role. Code-reuse attacks depend on code snippets that appear in the compiler-generated code and, thus, code generation is a key

```
1  move  $a0, $zero          # Move zero to $a0
2  lw    $ra, 0x24($sp)      # Load address for next gadget
3  jr    $ra                 # Jump to address at $sp + 4*0x24
4  addiu $sp, $sp, 0x28      # Delay slot: increment $sp
```

Figure 2.1: Code-reuse gadget in Mips libc

software-development stage to affect these attacks. Similarly, compilers may affect mitigations against side-channel attacks [24, 25, 26], and thus, compilation is the appropriate stage for security property preservation.

Code-Reuse Attacks

Code-reuse attacks depend on memory-corruption vulnerabilities in the program memory space, such as a buffer overflow. These vulnerabilities allow a malicious actor to insert data into the target program memory. To prevent direct attacker-introduced payload execution, executable-space protection prohibits the execution of code in writable memory, e.g., the stack.

Code-reuse attacks, such as return-to-libc, and advanced attacks, such as Return-Oriented Programming (ROP) [6, 27] and Jump-Oriented Programming (JOP) [28, 29, 30], may bypass executable-space protection defenses. The attackers use code snippets in known locations in the program memory of the target system to design the attack. These code snippets, so-called gadgets, typically end with a control-flow instruction, such as a return, a jump, or a call instruction. The last control-flow instruction allows the attacker to build a gadget chain by transferring the control from one gadget to the next. The gadgets that are available in the victim program memory, such as in dynamically-loaded libraries, may provide high expressibility to allow the attacker to hijack the system [6].

Figure 2.1 shows a code-reuse gadget found in the libc library of a Mips32 Debian Linux system. At line 1, the gadget moves value zero to register `$a0`. Then, it loads the address of the next gadget to the return-address register `$ra`. The attacker has ensured that this address resides to address `0x24($sp)`, where `$sp` is the stack pointer. At line 3, the code moves to the new gadget, *jr* `$ra`, and the last instruction is a delay slot[1]. The delay slot increases and stack pointer `$sp` by `0x28`. This last step is important for the attacker to move the attack payload forward to enable moving to the data of the next gadget. Such code sequence appear at the return points of functions and are very common in compiler-generated code.

Classic code-reuse attacks assume that the attacker has access to binary code identical to the victim code and designs a payload offline before attacking the victim system. More advanced attacks allow reading the memory during the attack. In

[1]Delay slots in Mips follow a branch instruction but execute before them and their purpose is to reduce the branch target estimation delay.

```
0x8014: move  $a0, $zero
0x8018: lw    $ra, 0x24($sp)
0x801c: jr    $ra
0x8020: addiu $sp, $sp, 0x28
```

(a) Variant 1

```
0x8010: lw    $t9, 0x24($sp)
0x8014: move  $a0, $zero
0x8018: nop
0x801c: jr    $t9
0x8020: addiu $sp, $sp, 0x28
```

(b) Variant 2

Figure 2.2: Two diverse implementations of the gadget in Figure 2.1 in Mips.

particular, JIT-ROP [31] dynamically reads the program memory, selects appropriate gadgets, and performs the attack. Similarly, Blind ROP (BROP) [32] achieves reading the memory using repeated steps, however, BROP may lead to system crashes. A different approach by Seibert et al. [33] achieves identifying the location of code snippets in the code using timing side-channel information. In particular, when the victim binary is not identical with the original binary, an attacker may use timing side-channel information to decipher the binary's diversification. The attacker records the execution time of different parts of the code to recognize how the code has been diversified. The advantage of this approach is that it does not lead to system crashes, however, it may take a long time, up to a week, to achieve the attack.

Code-Reuse Attack Mitigations

There are two main mitigation approaches against code-reuse attacks, Control-Flow Integrity (CFI) [34] and automatic software diversification [7]. The main idea of CFI is to ensure that the program executes legitimate control flow [35]. In this way, CFI restricts code-reuse attacks to only transfer control to legitimate control-flow targets, which reduces the power of these attacks[2]. The main disadvantage of CFI is that it often leads to high execution-time overhead [34]. An alternative mitigation against code-reuse attacks is automatic software diversification, which introduces uncertainty to the implementation of the code and, in this way, hinders a code-reuse attack that depends on the addresses of known gadgets. More generally, software diversification is a method to improve the fault tolerance of software systems [36, 37] and security [38, 39] in computing systems. Software diversification investigates code diversity at the level of algorithm implementation, library implementation [40, 41], memory layout (Adress Space Layout Randomization (ASLR)), and binary-level implementation [8, 9]. This dissertation concerns automatic fine-grained software diversification that generates program variants derived from the same source code but with different binary implementations.

Many code-reuse attacks depend on gadgets, which are code sequences that exist in the program memory. Typically, classic code-reuse attacks depend on the

[2]Other approaches, such as stack canaries, have similar effect on ROP attacks.

```
1 uint8 check_bit(uint8 pub, uint8 key) {
2   uint8 t = 0;
3   if (pub == key)
4     t = foo();
5   return t;
6 }
```

Figure 2.3: Program with secret-dependent branching in C

exact addresses of the gadgets in memory. Software diversification as a mitigation against code-reuse attacks changes the address of these gadgets and/or their implementation, aiming to reduce the probability of success of an attack. Figure 2.2 shows two versions of the gadget in Figure 2.1, where Figure 2.2a corresponds to the original gadget in Figure 2.1. The gadget in Figure 2.2b differs from the gadget in Figure 2.2a in three points, 1) there is an additional No-Operation (NOP) instruction at address `0x8018`, 2) instructions *move* and *lw* are swapped, and 3) the return address is loaded at register `$t9` instead of register `$ra`. Assuming that the attacker has used the gadget in Figure 2.2a to generate their payload, this payload may fail against a user using the code in Figure 2.2b. More specifically, an attacker that uses the gadget in Figure 2.2a will instruct the previous gadget to jump to address `0x8014`, namely the beginning of the gadget. However, at address `0x8014`, the diversified gadget (Figure 2.2b) moves the value of zero to register `a0`, but it does not move the attacker-controlled address to `$t9`. Hence, the processor will not transfer the control flow to the next gadget to finalize the attack. This scenario leads, with a high probability, to attack failure. Additional diversification approaches like function shuffling or basic-block shuffling may increase the entropy of this diversified example.

Side-Channel Attacks and Mitigations

Side-channel attacks use side-channel information, such as the execution time or the power consumption of a program, to extract information about valuable program values. Side-channel attacks constitute a severe threat to cryptographic algorithms. The security of cryptographic algorithms often depends on values that should remain secret, such as symmetric or asymmetric keys. Using side-channel information, an attacker may extract information about these keys to break cryptographic security. Side-channel attacks have been successful against popular cryptographic algorithms, such as DES, AES, and RSA [10, 11, 12].

Timing Side-Channel Attacks

Timing side-channel attacks [12] measure the execution time of the program to extract information about program values. The attacker may perform the attack

```
1  uint8 sbox_get(uint8 *pub, uint8 key) {
2    return pub[key];
3  }
```

Figure 2.4: Program with secret-dependent memory operation

remotely via the network [42] or locally when the victim and the attacker share the same hardware [43]. Timing attacks may extract information about a secret value when this value affects the program execution.

In many implementations of cryptographic algorithms, the execution time may depend on the value of the encryption/decryption key. For example, consider function `check_bit` in Figure 2.3. Assume that the value of `pub` is known to the attacker, whereas `key` contains secret information. If these two values are equal, the program executes function `foo()`, whereas otherwise, the `check_bit` function returns immediately. Hence, if the two values are equal, then the execution of function `check_bit` takes longer time than otherwise. Therefore, an attacker that measures the execution of this function may distinguish the difference between the execution time, e.g. by controlling the value of `pub` and, subsequently, extract one bit of information about the value of `key`.

Another timing vulnerability that appears in cryptographic implementations is secret-dependent memory operations. Figure 2.4 shows function `sbox_get`, which takes two inputs, a public array, `pub`, and a secret value, `key`. The function returns the element of `pub` at index `key`. Here, the source of the leakage is the cache hierarchy, which aims at providing recently-accessed address regions faster. In particular, cache memories have low data access latency and store recently accessed memory blocks for faster access in future memory requests by the processor. The cache stores these blocks based on their address in memory, which depends on the array index, `key`, in our example. Upon a memory request in the cache, if the relevant cache line is full, the cache replaces old data with the new data. An attacker may take advantage of the cache hierarchy to extract information about secret values. For example, an attacker may fill a shared cache with their own data before the victim runs their code. Subsequently, the attacker measures the access time of their data to infer the memory access patterns of the victim (Prime+Probe) [44].

Other sources of timing vulnerabilities include variable-latency instructions with secret operands, such as division and multiplication instructions in some architectures. In general, when the execution time depends on secret values that should remain unrevealed, execution time may leak information about these secret values.

Timing Side-Channel Mitigations

In the research literature, there are diverse mitigation approaches against timing side-channel attacks. This dissertation concerns two mitigation approaches: constant-time programming and constant-resource programming.

```
1 uint8 check_bit(uint8 pub, uint8 key) {
2   uint8 t; sint8 m;
3   t = foo();
4   m = -(pub == key);
5   return (t&m | 0&~m);
6 }
```

Figure 2.5: Constant-Time Program from Listing 2.3 in C

Constant-Time Programming: The constant-time programming discipline is a set of programming guidelines that aim at removing secret-dependent timing variations. The constant-time discipline converts secret-dependent branch instructions and memory accesses to constant-time equivalent. In addition, some approaches consider secret-dependent variable-latency instructions, such as division and multiplication, when their operands are secret values.

Figure 2.5 shows a constant-time version of function `check_bit` in Figure 2.3. This implementation starts by executing function `foo()` (line 3), regardless of the result of the comparison between the two input values. Then, the code stores the negation of the result of the comparison between the two input values in the signed variable `m` (line 4). That is, if the result of the comparison is true or `000000001` in binary encoding, then variable `m` is minus one or `11111111` in binary encoding. Similarly, if the result of the comparison is false or equal to zero, then minus zero is zero, which leads to `00000000` in binary encoding. The return result (line 5) is either equal to `t`, when the value of `m` is `11111111`, or `0`, when `m` is `00000000`. Hence, in the constant-time reimplementation of Figure 2.3, the execution time does not depend on the secret value, instead, it is constant.

Constant-time implementations may contain complex logic-operation code that is often difficult to write, read, and debug. Hence, verification approaches are important for guaranteeing the correctness of these implementations. Furthermore, compilers may break such implementations, for example, by converting logical operations back to a secret-dependent branch [24].

Constant-Resource Policy: The constant-time programming discipline often leads to complex code. Another alternative mitigation approach against timing side-channel attacks is the constant-resource policy [45]. In contrast to the constant-time policy, the constant-resource policy does not require the absence of secret-dependent branches and memory operations, instead it requires that the program uses the same resources for different secret values [46]. In particular, the constant-resource policy allows secret-dependent branches when both branch directions lead to the same execution time.

Figure 2.6 shows a constant-resource version of function `check_bit` in Figure 2.3. At line 3, the implementation compares the two input values. If the input values are equal, the implementation calls function `foo()` and stores the result in variable

```
1 uint8 check_bit(uint8 pub, uint8 key) {
2   uint8 t = 0, _t;
3   if (pub == key)
4     t = foo();
5   else
6     _t = foo();
7   return t;
8 }
```

Figure 2.6: Balanced Constant-Resource Program from Listing 2.3 in C

`t` (line 4). If the two input variables are not equal, the code calls function `foo()` and stores its result in an unused variable `_t` (line 6). The return result of this function is `t`, which is either `0` or the return value of function `foo()`. The idea of this transformation is that both branch directions take the same execution time[3].

This implementation is easier to read and more similar to the original implementation than the constant-time equivalent in Figure 2.5. However, there are two main compilation challenges of such implementations. First, dead-code elimination passes may remove the functionally unused call to function `foo()` in the `else` branch. Second, timing variations between the two branches may depend on different microarchitectural features, such as instruction latencies, branch prediction, and memory accesses. The latter may lead to a longer execution time for either of the branches. These challenges make compiler-based approaches that consider an accurate cost model valuable to guarantee that different execution paths use the same resources.

Power Side-Channel Attacks

Power side-channel attacks measure the power consumption during the execution of a program to exploit the victim system or extract security-critical information. For power side-channel attacks, any value transitions in hardware, such as hardware registers, the memory, or the memory bus, may reveal information about these values. When a program manipulates secret information, the attacker may disclose this information by recording the power consumption during the program execution. Figure 2.7 shows function `xor`, which takes two values as input, `pub`, which is known and/or controlled by the attacker and `key`, which is a secret value unknown to the attacker. The function returns the exclusive-OR operation of the two input values. These operations may affect the device's power consumption because the processor manipulates the secret value with a public value in the hardware.

A typical leakage model for power attacks is the Hamming model. A data word consists of m bits, each of which takes a value from $[0, 1]$. We can write a data

[3]In general, the actual timing of each of the branches depends also on microarchitectural features including, branch decisions, instruction latencies, and memory accesses

```
1  uint32 xor(uint32 pub, uint32 key) {
2    uint32 t = pub ^ key;
3    return t;
4  }
```

Figure 2.7: Exclusive-OR implementation in C

word in form $D = \sum_{i=0}^{m-1} d_i 2^i$, where d_i is the value of the binary encoding at the i_{th} position. The Hamming weight of a data word corresponds to the number of bits that are one, i.e. $H(D) = \sum_{i=0}^{m-1} d_i$. Many works assume that the data leakage through a power side channel depends on the number of bits switching from one to zero or vice versa at a given time. Hence, the data leakage at the transition of one hardware variable from D_1 to D_2 equals $H(D_1 \oplus D_2)$, where $\oplus$ is the exclusive-OR operator. The transition between these values happens at distinct time points, for example, at the positive or negative clock edge in an electronic device [11]. The data leakage may happen at different parts of the processor, for example, the memory bus, the hardware registers, the memory cells, and more [11, 13, 47]. According to Papagiannopoulos and Veshchikov [13], the easiest to exploit transitional leakages are hardware-register and memory-bus reuse in an AVR processor.

Power side-channel attacks require the attacker to have local access to the victim device and have equipment such as an oscilloscope to record the power consumption of the target algorithm at the victim device. Simple Power Analysis (SPA) is a technique to make direct observations on the power trace of an algorithm in time. SPA allows the attacker to extract information when indirect branches, value comparisons, multiplication operations, and exponentiation operations depend on secret values. SPA allows secret information extraction from multiple DES procedures [10]. Differential Power Analysis (DPA) may distinguish smaller variations in the power traces that may be too small to distinguish using SPA. DPA records the power traces of multiple executions of the algorithm. By observing the algorithm's output, e.g., the ciphertext, the attacker determines the values of secret data, such as cryptographic keys [10]. Correlation Power Analysis (CPA) [11] uses the correlation factor between the hamming distance of data and the measured power to determine the relationship between a guessed value and the actual measurement. This analysis depends on the Hamming distance model and is as powerful as DPA. In recent years, power attacks based on deep learning has enabled more powerful attacks [48].

Power Side-Channel Mitigations

Power side-channel attacks do not affect the program execution and are, therefore, difficult to detect. One mitigation approach against power side-channel attacks is software masking. This mitigation aims at randomizing the secret values using

```
1  uint32 sec_xor(uint32 pub, uint32 key, uint32 mask) {
2    uint32 mk = mask ^ key;
3    uint32 t  = mk ^ pub;
4    return (t,mask)
5  }
```

Figure 2.8: Exclusive-OR with software masking

```
1  sec_xor(r0 ← pub, r1 ← key, r2 ← mask) {
2    r2 ← r2 ^ r1;
3    r0 ← r2 ^ r0;
4  }
```

Figure 2.9: Exclusive-OR with software masking in machine code

randomly generated values at every program execution. Software masking aims at statistically hiding the secret information.

Software masking depends on finite field theory and uses the exclusive-OR operation as the addition operation in $GF(2^n)$. Figure 2.8 shows function `sec_xor`, which is an implementation of Figure 2.7 using software masking. First, this function randomizes the secret value `key` using a newly introduced randomly generated value `mask` (line 2). Then, the function uses the randomized key value, `mk`, to perform an exclusive-OR operation with `mask` (line 3). Finally, function `sec_xor` returns the final value and variable `mask`, which is necessary for retrieving the final result, namely `(mask ^ key ^ pub) ^ mask` is equal to `key ^ pub`.

Although the function implementation in Figure 2.8 randomizes the secret before interacting with the public value, hardware interactions of values may introduce secret-dependent power dependencies that appear in the power traces. These interactions occur when hardware registers, the memory bus, or memory cells transition from one value to another. For example, when a hardware register takes a new value or when a new value is transferred via the memory bus to the main memory, may result in a transition. Assuming that transitions from one to zero and from zero to one result in a similar power consumption change, the hardware's total power change depends on the hamming distance between the old value and the new value at the transition. Figure 2.9 shows an implementation of the masked algorithm in Figure 2.8 using hardware registers. At line 2, register `r2`, which holds value `mask`, transitions to the value of `r2 ^ r1`, which holds value `key ^ mask`. The leakage from this transition is equal to `mask ^ (key ^ mask)` or `key`, which is a secret value. Hence, this hardware implementation of the masked code leads to transitional leaks due to register reuse. Similarly, other hardware value transmissions may lead to transitional leaks of secret values.

Many transitional leaks depend on compiler-generated machine code, which determines hardware register assignment and instruction order. Furthermore, some mitigations against these transitional leaks result in high performance overhead [13]. Therefore, compiler-based approaches may provide opportunities to mitigate these leakages at a reduced performance overhead.

2.2 Constraint Programming

Constraint Programming (CP) is a method to solve or optimize combinatorial problems. Compared to other combinatorial solving or optimizing approaches, such as Boolean Satisfiability (SAT) and linear programming, CP enables larger flexibility with regard to the domain of variables and the type of constraints. The main strength of CP is its ability to exploit substructures in combinatorial problems and has been particularly successful in solving scheduling, resource allocation, and rectangle packing problems [49].

CP solving usually consists of two parts, 1) modeling, where the user models the problem as a finite set of variables and constraints, and 2) solving, where the solver attempts to find solutions, i.e. variable assignments that satisfy all constraints, to the problem. A Constraint Satisfaction Problem (CSP) models a problem for which we need to find one or multiple solutions to the problem. Constraint Optimization Problems (COPs) include an *objective* function, and the goal is to find the solution(s) that optimize(s) this objective function.

Modeling

In CP, a problem is modeled as a finite set of variables, V, that takes values from a finite set, U, and a set of constraints, C, among the variables in V. Typical variable domains include integer and Boolean sets. CP solvers provide different constraint implementations among different variable types.

Problem Modeling: The first step in CP is problem modeling. Modeling is important because modeling decisions affect the solving time of the problem.

A typical example of a combinatorial problem is the eight-queen problem, namely the problem of placing eight (chess) queens on a chessboard, so that they do not threaten each other. There are (at least) two ways to model the eight-queens problem. One way to model this problem is to use eight variables, q_i, one for each queen. Each variable q_i corresponds to the i_{th} column (or row) of the chessboard, and its value corresponds to the position in the row (or column). The variable domain, in this case, will be set $\{1, 2, ..., 8\}$. A different way to model the eight queens problem is using one variable, c_i^j for each chessboard position. In this way, we have 64 variables with domain, $\{0, 1\}$. A variable takes value one when a queen is present on the corresponding cells and zero otherwise. These two different modeling approaches may lead to different solving times due to different properties

of the solver and individual constraint implementations. Therefore, constraint modeling is an essential step in constraint solving.

Constraints: Constraints are important for modeling a constraint problem. Different constraints affect the efficiency and expressiveness of the solving procedure. Constraints that involve three or more variables are also called *global* constraints [49] and often provide improved efficiency. Global constraints constitute one of the key strengths of CP.

The first modeling approach of the eight-queens example may use a global constraint, `all-different`$(q_1, ..., q_8)$ [50], to make sure that all variables differ from each other; namely, they do not share the same row/column. The `all-different` constraint has improved efficiency over a set of disjunctive constraints $\forall i, j \in \{1, ..., 8\}.q_i \neq q_j$.

The second modeling approach may use a global linear constraint $\forall i \in \{1, ..., 8\}. \sum_{j \in \{1,...,8\}} c_i^j = 1$ to ensure that every row accommodates only one queen. Note that this constraint is an efficient way to model that one and only one queen appears in a row or $\forall i \in \{1, ..., 8\}.\exists j \in \{1, ..., 8\}. \left(c_i^j = 1 \wedge \forall k \in \{1, ..., 8\} - \{j\}.c_i^k = 0\right)$.

Solutions: The solutions to the problem are the variable assignments that satisfy all constraints.

Example 1 *Give a CSP $P = \langle V, U, C\rangle$, where $V = \{x, y, z\}$, $U = \{1, 2, 3, 4\}$, and $C = \{x > 1, x + y = z\}$, the solutions to the problem are: $sol(P) = \{\langle 2, 1, 3\rangle, \langle 2, 2, 4\rangle, \langle 3, 1, 4\rangle\}$*

In some problems, all solutions are not equivalent, and the application requires a solution that, e.g., maximizes or minimizes an *objective* function, O. In this case, the goal of the solver is to find the *optimal* solution to the problem.

Example 2 *CSP P may be extended to a COP, $P' = \langle V, U, C, O\rangle$, with an optimization function $O =$ `maximize` x. Then, the solution that we are looking for is $sol(P') = \{\langle 3, 1, 4\rangle\}$.*

For many problems, it is sufficient to find one (optimal or not) solution, whereas, for other problems, it is essential to find a set of diverse solutions [51]. Example applications include automatic test generation [52], finding alternative optimal solutions in process plant layout optimization [51], and solving complex constraints by generating multiple solutions and then verifying the suitability using more exact methods [53]. Defining the meaning of the difference between solutions is essential in this context. To achieve this, we define function δ that takes two solutions and returns the difference between these solutions.

Example 3 *For problem P, we may define the function $\delta(s, s') = |s_x - s'_x| + |s_y - s'_y| + |s_z - s'_z|$. Then, the distance between all three pairs of solutions is $\delta(s_i, s_j) = 2$, $i, j \in \{1, 2, 3\}$.*

Solving

Given a CSP or a COP, the solving process aims at finding feasible solutions. Typically, solving in CP is an iterative procedure and consists of two main steps, 1) propagation, which reduces the variable domains based on the constraints, and 2) search, which sets the value of one variable from its domain in each step.

Propagation: Constraint propagation is the procedure that reduces the variable domains based on the problem constraints. Often the propagation process considers one constraint at a time and is repeated until reaching a fixpoint.

Example 4 *CSP* $P = \langle V, U, C \rangle$*, where* $V = \{x, y, z\}$*,* $U = \{1, 2, 3, 4\}$*, and* $C = \{x > 1, x+y = z\}$*, has two constraints* $C_1 = x > 1$ *and* $C_2 = x+y = z$*. Propagation of* C_1 *results in* $x \in \{2, 3, 4\}, y \in \{1, 2, 3, 4\}, z \in \{1, 2, 3, 4\}$*. Then, propagation of* C_2 *results in* $x \in \{2, 3\}, y \in \{1, 2\}, z \in \{3, 4\}$*, which is a fixpoint.*

Search: Propagation is usually not sufficient to solve a problem. After propagation has reached a fixpoint, the solver applies search to decompose the problem into simpler subproblems. In particular, the solver selects one variable and splits its domain, often in two parts.

Example 5 *In our problem, P, search may split the previous fixpoint (*$x \in \{2, 3\}, y \in \{1, 2\}, z \in \{3, 4\}$*) into two subproblems, one for* $x = 2$ *and one for* $x = 3$*.*

Subsequently, the solver applies propagation to each subproblem to find solutions and repeats the search step until it finds one solution, all solutions, or a specific solution.

Example 6 *For the first branch,* $x = 2$*, propagation returns fixpoint* $x \in \{2\}, y \in \{1, 2\}, z \in \{3, 4\}$*, which is not a single solution. Thus, we need to apply search again for* $y = 1$ *and* $y = 2$*, which gives two solutions* $x \in \{2\}, y \in \{1\}, z \in \{3\}$ *and* $x \in \{2\}, y \in \{2\}, z \in \{4\}$*, respectively.*

For the second branch, $x = 3$*, we get directly one solution after propagation:* $x \in \{3\}, y \in \{1\}, z \in \{4\}$*.*

Deciding which variable to branch on and how to split the value space of the selected variable at an invocation of search may affect the efficiency of the solving procedure and is an important design decision. These branching schemes are called search heuristics. Typical search heuristics include *smaller value first*, *variable with the smallest value set first*, and more.

Branch-and-bound search is an approach to solving COP problems. First, the algorithm finds one solution. Then, the algorithm adds a new constraint to the problem that requires the following solution to be better than the current one. The solver continues until there are no better solutions.

Example 7 *For P' that we used before, with $O =$ `maximize` x, we have the first propagation fixpoint as: $x \in \{2,3\}, y \in \{1,2\}, z \in \{3,4\}$. Assume we first find solution $\langle 2,1,3\rangle$. Then, branch-and-bound adds constraint $x > 2$, which leads to the optimal solution, $\langle 3,1,4\rangle$.*

Apart from the classic search heuristics, there are other search procedures called metaheuristics [54]. In this dissertation, we use Large Neighborhood Search (LNS) [55], a form of local search that is consistent with CP. LNS is often used for solving optimization problems. After finding the first solution, LNS uses part of this solution to find a better solution. To do that, LNS *destroys* parts of the solution (assignments to variables) and then tries to find other solutions that improve the objective function.

Example 8 *In our example, P', with $O =$ `maximize` x, we have the first propagation fixpoint as: $x \in \{2,3\}, y \in \{1,2\}, z \in \{3,4\}$. Assume we first find solution $\langle 2,1,3\rangle$. Then, LNS destroys variables x and z, and adds an optimization constraint $x > 2$ and after propagation we have: $x \in \{3\}, y \in \{1\}, z \in \{4\}$, which is the optimal solution.*

2.3 Compiler Backend

Compilers are essential components in the software development chain. Typical general-purpose compilers take as input a program written in a high-level language and translate it to a low-level language or machine code. Conventional compilers, such as LLVM [56] and GCC [57], consist of a series of analysis and transformation passes that aim to improve the code performance, reduce the code's size, or minimize the energy consumption.

Compiler front- and middle-end passes perform high-level transformations such as loop unrolling, dead-code elimination, and expression rewriting, whereas compiler back-end passes are responsible for target-processor-related transformations. At the compiler backend, there are three main transformations, 1) instruction selection, where machine instructions replace abstract instructions, 2) instruction scheduling, which decides the order of the instructions in the final code, and 3) register allocation, which assigns virtual registers to hardware registers and memory. These transformations are very important for the quality of the generated code in the hardware and are increasingly important in architectures that require significant effort from the compiler, such as static multiple-issue architectures. Such architectures typically require the compiler to schedule multiple instructions to different processing units statically. However, the compiler backend transformations are known to be combinatorial problems, where finding the optimal hardware code implementation may take exponential time. Instead, many compilers use heuristics that find efficient but not optimal solutions.

Combinatorial Compiler Backend

For compiler-demanding architectures or performance-critical functions, there are combinatorial compiler-backend approaches to find optimal low-level implementations [58, 59]. These approaches use an abstract processor model and represent the quality of each solution in the form of a cost function. Subsequently, combinatorial compiler approaches generate code that optimizes this objective function. In combinatorial optimization, an optimal solution corresponds to a solution that maximizes or minimizes the objective function. Combinatorial models do not exclude the presence of multiple optimal solutions.

Unison, a recent work, has shown the benefits of unifying multiple compiler passes to generate highly optimized code. In particular, instruction scheduling and register allocation are strongly interdependent problems, and thus, modeling both problems together improves the quality of the generated code [5]. Unison is the first practical combinatorial compiler-backend approach, and SecOpt is based on Unison. In particular, SecOpt extends Unison to consider security properties.

Modeling: Typically, compiler-backend passes process the program in Static Single Assignment (SSA) form, where every variable is assigned a value only once. A combinatorial compiler models a program as a set of basic blocks B, i.e. pieces of code with no branches apart from the exit of the block. Each basic block contains a number of optional operations, $o \in \mathit{Operations}$, that may be *active* or not. An active operation appears in the generated code, while an inactive operation is omitted. These optional operations enable transformations that are necessary for register allocation and instruction scheduling. Ins_o denotes the set of hardware instructions that implement operation o. Each operation includes a number of operands $p \in \mathit{Operands}$, each of which may be implemented by different, equally-valued temporaries, $t \in \mathit{Temps}$. Temporaries represent an infinite number of virtual registers, which the solver assigns to a hardware register or a memory location in the stack. Alternatively, a temporary may not be alive.

Fig. 2.10 shows a simplified version of the constraint-based compiler backend model for Fig. 2.7. Temporaries `t0` and `t1` contain the input arguments `pub` and `key`, respectively. Copy operations (`o2, o3, o5`) enable copying program values from one register to another (or to the stack) and are critical for flexibility in register allocation. Operation `o2` allows the copy of value `pub` from `t0` to `t3`. In the final solution, a copy operation may not be active (shown by the dash in the set of instructions: `[-, copy]`). The `xor` operation (`o4`) takes two operands, and each of these operands may use equally valued temporary variables, e.g., `t1` and `t4`.

Objective Function: Combinatorial compiler backends use an objective function that is based on the target processor characteristics to generate optimized code. Unison's objective function optimizes metrics such as *code size* and *execution time*. Unison captures these goals in a generic objective function that sums up the

```
o1: in [t0 ← pub, t1 ← key]
o2: t2 ← [-, copy] t0
o3: t3 ← [-, copy] t1
o4: t4 ← xor [t1,t3] [t0,t2]
o5: t5 ← [-, copy] t5
o6: out [t6 ← [t4,t5]]
```

Figure 2.10: Exclusive-OR operation

weighted cost of each basic block:

$$\sum_{b \in B} \mathit{weight}(b) \cdot \mathbf{cost}(b),$$

where $\mathbf{cost}(b)$ for basic block b is a variable, which estimates the cost of a specific implementation of the basic block, and *weight* is a constant value that represents the contribution of the particular basic block to the total cost. For execution-time optimization, Unison uses statically extracted basic-block frequencies to estimate the contribution of each basic block. This cost model is accurate for predictable hardware architectures. The accuracy of the cost model reduces in the presence of advance microarchitectural features, such as cache hierarchy, dynamic branch prediction, and/or out-of-order execution.

Solving: After modeling the problem, the constraint solver attempts to optimize the problem using scheduling constraints. Unison uses structural decomposition and advanced search strategies to find the optimal or a good solution efficiently.

Figure 2.11 shows a solution to the program in Figure 2.10. Temporaries `t0` and `t1` are assigned to hardware registers `r0` and `r1`, respectively, due to the calling conventions of the target architecture. Temporary `t2` is assigned to `r2`, and the operation copies the value of `r0` to `r2`. Operation `o3` is not active, and thus, temporary `t3` is not live. Temporary `t4` is assigned to register `r0`. Operation `o5` is also not active, and temporary `t5` is not live. Finally, temporary `t6` is assigned to the return register `r0`. This solution is suboptimal, instead, the optimal solution (see Figure 2.12) deactivates all `copy` operations.

```
o1: in [t0:r0 ← pub, t1:r1 ← key]
o2: t2:r2 ← copy t0:r0
o4: t4:r0 ← xor t1:r1 t2:r2
o6: out [t6:r0]
```

Figure 2.11: Solution of exclusive-OR operation

```
o1: in [t0:r0 ← pub, t1:r1 ← key]
o4: t4:r0 ← xor t1:r1 t0:r0
o6: out [t6:r0]
```

Figure 2.12: Optimal solution of exclusive-OR operation

Transformations: Unison enables the following transformations: 1) hardware register assignment, 2) register copying, 3) memory spilling, 4) constant rematerialization, 5) instruction order, and 6) NOP insertion. Hardware register assignment maps a hardware register to an operand, register copying copies an operand from one hardware register to another, and memory spilling allocates a memory slot in the stack to store an operand. Constant rematerialization allows re-running an operation, like value loading, instead of copying its result. SecOpt may also alter the instruction order as long as there are no data dependencies or insert NOP instructions by delaying the issue cycle of an operation.

Chapter 3

Approach and Methodology

3.1 Secure-by-Design Optimization (SecOpt)

This section presents our approach, SecOpt, to generate secure and optimized code by design. SecOpt implements the following mitigation approaches: 1) fine-grained software diversification, 2) software masking, and 3) preservation of the constant-resource property.

Overall, there are three main advantages of SecOpt compared to other approaches, 1) performance awareness, 2) composability, and 3) a formal definition of mitigations in the form of constraints. SecOpt inherits an accurate cost model from Unison [5], which allows control over the performance overhead of the generated code. In particular, SecOpt may generate optimal or near-optimal code with a known performance overhead that is based on the cost model. An important advantage of SecOpt is combining different security mitigations. More specifically, SecOpt allows combining fine-grained software diversification and software masking or the constant-resource property to ensure the preservation of the combination of these properties. Moreover, many conventional compiler-based mitigation approaches do not provide formal guarantees that the intended properties hold in the generated code. SecOpt defines the target mitigations in the form of constraints,

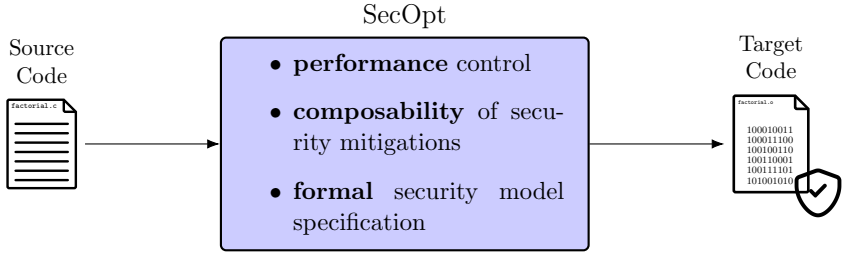

Figure 3.1: High-level view of SecOpt

providing guarantees about preserving the intended properties. However, SecOpt's security analysis, transformations, and the underlying constraint solvers are not verified. Verifying these compilation stages is part of future work for Unison [5] and SecOpt. Code verification uses formal methods to prove software properties or identify violations of these properties in the code. To ensure that the generated code against timing side channels satisfies the constant-resource property, we use external static-analysis tools that output the over- and under-approximation of the execution time. This verification stage increases our trust in SecOpt's code generation. Furthermore, we investigate the constant-time property in WebAssembly programs using relational symbolic execution, which generalizes symbolic execution to prove relational properties [60].

Figure 3.1 shows a high-level view of SecOpt. SecOpt takes as input a program in a high-level language, such as C and C++, and outputs a secure binary-code implementation. The following sections describe the approach of this dissertation to protect binary code against code-reuse and side-channel attacks.

Fine-Grained Software Diversification

To enable code diversification, SecOpt uses the transformation search space that the constraint model allows. In particular, there are multiple solutions to the constraint model of SecOpt that satisfy the model of the program semantics, the target processor model, and the low-level transformations. These solutions may be optimal according to the cost model or suboptimal. To generate diverse solutions, we need to define the a *distance measure*, δ, which is a constraint between two alternative solutions to the problem and measures how *different* these solutions are. The problem definition is the following:

Definition 1 *Diverse Code Generation: Consider a program p and a set, S, of program implementations of p, $p_i \in S$, which are functionally equivalent ($\sim$) with the original program, $\forall p_i \in S.\ p \sim p_i$, and each other, $\forall p_i, p_j \in S.\ p_i \sim p_j$. Furthermore, these program implementations differ from each other based on a distance function δ, namely: $\forall p_i, p_j \in S.\ \delta(p_i, p_j)$.*

In this definition and later in this section, we define δ as a predicate that is true when the compared solutions are different and false otherwise. This simplification implies that our model assumes that two program implementations are different when they differ by one model variable, which decides the register assignment or the program schedule. Note that this constraint enforces that the program implementations are different but does not restrict how different these implementations are. The actual implementation of our approach allows control over the value of the distance function to enforce more diversity among the solutions.

To generate highly diverse and optimized code, SecOpt uses a local-search method, LNS, to navigate in the program's search space around the optimal solution. More specifically, SecOpt finds first the optimal solution, y_{opt}, according to

the cost model and then uses LNS to find alternative solutions around this optimal solution. Initiating the search starting from the optimal solution allows the solver to locate highly optimized solutions quickly. To control the performance overhead in the generated variants, we introduce a constraint C_{opt} that restricts the cost function to have at most $g\%$ overhead. At the same time, SecOpt introduces a distance measure that forces the solutions to differ from each other. In particular, SecOpt uses an iterative algorithm as follows:

```
1  y ← y_opt;                      // Start with optimal solution
2  S ← {y_opt};                    // Add optimal solution to S
3  C' ← C ∪ {Δ(y_opt), C_opt};     // Add constraints
4  while cont_cond()               // Iterate until limit
5    y ← solve_LNS(y, C');         // Find next solution
6    S ← S ∪ {y};                  // Add new solution to S
7    C' ← C' ∪ {Δ(y)};             // Add diversity constraint
8  return S                        // Return set of diverse solutions
```

At line 1, the algorithm copies the optimal solution to the current solution to proceed in the iteration. At line 2, the algorithm adds the optimal solution to the set of solutions, S. Then, at line 3, the algorithm updates the set of constraints with two constraints. First, distance constraint $\Delta(y_{opt})$ ensures that all future solutions to the problem y' will differ from y_{opt}, namely $\delta(y', y_{opt})$. The second constraint C_{opt} restricts the solutions to have at most $g\%$ performance overhead. Lines 4 to 7 implement the iterative algorithm that proceeds until it reaches a condition, such as a time limit or the maximum number of variants (line 4). At line 5, the algorithm takes the previous solution and finds a new solution using LNS. Subsequently, the algorithm inserts the new solution to set S (line 6), and finally, the algorithm updates the set of constraints C' so that the future solutions are different from the newly found solution.

Automatic fine-grained software diversification is effective against code-reuse attacks, however, software diversification approaches against side-channel attacks lead to a large overhead [14]. Instead, to secure the code against side-channel attacks, SecOpt preserves software mitigations against side-channel attacks.

Optimizing Side-Channel Mitigations

The underlying constraint-based compiler backend of SecOpt generates highly-optimized binary code. However, these optimal solutions do not necessarily satisfy security constraints. Enforcing the generation of secure solutions requires extending the constraint model to include security constraints. The security properties that SecOpt implements are software masking against power side channels and the constant-resource property against timing side channels. The selected mitigations depend on the underlying compiler-backend's transformation space and our goal to generate highly optimized code for resource-constrained devices. In particular,

both software masking and constant-resource programming introduce performance overhead that a combinatorial approach may reduce [13, 61].

To investigate the feasibility and adequacy of a secure optimizing approach, we express each mitigation as part of the constraint model of the underlying constraint-based compiler backend. Definition 2 defines the problem statement for secure code generation.

Definition 2 *Secure Constraint-Based Optimization: Given a constraint problem* $P = \langle V, U, C, O \rangle$ *that describes a constraint-based compiler backend in CP, we define constraints* C_{sec}*, such that problem* $P_{sec} = \langle V, U, C \cup C_{sec}, O \rangle$ *satisfies solutions that mitigate the relevant vulnerabilities.*

An important initial step for generating the input data for the security constraints, C_{sec}, is security analysis of the code. This analysis takes as input a security policy that defines which program variables are secret, public, or random to identify possible vulnerabilities in the program.

Software Masking

SecOpt generates code that protects against transitional leakages due to hardware-register reuse and memory-bus reuse. Before generating the constraints, SecOpt performs static analysis to identify possible sources of leaks in the code. In particular, SecOpt adapts the type-inference algorithm from Wang et al. [25] to extract a set of program variables and operations that may lead to transitional leakages. This analysis returns a set of pairs of program variables, $RPairs$, and a set of pairs of memory operations, $MPairs$. Each pair in $RPairs$ leaks secret information when a hardware register transitions from one value to another. Analogously, a pair of operations in $MPairs$ leaks secret information if the two operations write to the memory bus subsequently. The constraint model that preserves software masking consists of constraints that use $RPairs$ and $MPairs$ to restrict the register allocation and instruction scheduling of the code generation. In particular, SecOpt extends the compiler-backend constraint model with the following set of constraints to protect against register-reuse leakages:

`conflict_rassign(`$RPairs$`):`
$\forall \mathtt{t}_1, \mathtt{t}_2 \in RPairs.\ r(\mathtt{t}_1) = r(\mathtt{t}_2) \implies \neg \mathtt{subseq}(\mathtt{t}_1, \mathtt{t}_2)$

Constraint `conflict_rassign` implies that if the temporaries in a pair of program variables that appears in $RPairs$, $\mathtt{t}_1, \mathtt{t}_2$, are assigned to the same register, $r(\mathtt{t}_1) = r(\mathtt{t}_2)$, then they should not be assigned subsequently (`subseq`). Here, constraint (`subseq`) considers both directions, namely $\mathtt{t}_1$ is assigned to $r(\mathtt{t}_1)$ immediately before $\mathtt{t}_2$ is assigned to the same register and vice versa.

Figure 3.2 shows one vulnerable (Figure 3.2a) and one secure (Figure 3.2b) implementation of the code in Figure 2.7. In both figures, the input variables, `pub`, `key`, `mask`, are stored in registers `r0`, `r1`, and `r2`, respectively. In these figures, we denote public values as (`p`), secret values as (`s`), and random values as (`r`). The

```
1  @ r0: pub (p), r1: key (s), r2: mask (r)
2  eors r2, r1 @ r2:r->r^s
3  eors r0, r2 @ r0:p->p^s^r
4  bx   lr
```

(a) Insecure

```
2  eors r1, r2 @ r1:s->s^r
3  eors r0, r1 @ r0:p->s^r^p
4  bx   lr
```

(b) Secure

Figure 3.2: Two program implementations of Figure 2.7 for ARM Cortex M0

```
1  @ r0: [pub] (p), r1: [key] (s), r2: [mask] (r), r3: [res]
2  ...
3  ldr  r1, [r1] @ M:p->s,r1:p->s
4  ldr  r2, [r2] @ M:s->r,r2:p->r
5  eors r2, r1   @ r2:r->r^s
6  ldr  r0, [r0] @ M:r->p,r0:p->p
7  eors r2, r0   @ r2:r^s->r^s^p
8  str  r2, [r3] @ M:p->r^s^p
9  ...
10 ...
11 ...
```

(a) Insecure (LLVM)

```
2  ...
3  ldr  r2, [r2] @ M:p->m,r2:p->m
4  str  r2, [sp] @ M:m->m
5  ldr  r2, [r1] @ M:m->k,r2:m->k
6  ldr  r1, [sp] @ M:k->m,r1:p->m
7  eors r2, r1   @ r2:k->k^m
8  ldr  r0, [r0] @ M:m->p,r0:p->p
9  eors r0, r2   @ r0:k^m->k^m^p
10 str  r0, [r3] @ M:p->m^k^p
11 ...
```

(b) Secure (SecOpt)

Figure 3.3: Two program implementations of the equivalent of Figure 2.7 using pointers for ARM Cortex M0

comments next to the code denote the value transitions (v_{old}->v_{new}) in a register, e.g. *r0*. The first instruction at line 2, *eors*, takes two operands r2 and r1, performs the exclusive-OR, and writes the result in register r2 (two-address instruction). This operation implies that there is a value transmission in register r2, from value mask to value mask ^ key, which leads to a hamming-distance leakage (mask ^ key) ^ mask, which is equal to key(circled in red). This leakage implies that the implementation leaks information about value key to a power side-channel attacker. The rest of the code does not lead to any leaks. To generate a secure implementation for the code in Figure 2.7, SecOpt changes the operand order of the *eors* instruction, as shown in Figure 3.2b (line 2). The new implementation leads to a leak of value mask, which is random.

Similarly, we add the following constraints to protect against memory-bus sharing leakages:

$$\texttt{conflict\_order}(MPairs): \quad \forall \texttt{o}_1, \texttt{o}_2 \in MPairs.\ \neg \texttt{msubseq}(\texttt{o}_1, \texttt{o}_2)$$

Constraint conflict_order implies that two operations $\texttt{o}_1, \texttt{o}_2$ should not be scheduled subsequently (msubseq). Constraint (msubseq) considers both directions, namely, $\texttt{o}_1$ before $\texttt{o}_2$ and vice versa.

Figure 3.3 shows one vulnerable implementation (Figure 3.3a) generated by

Unison and one secure implementation (Figure 3.2b) generated by SecOpt. This implementation is a variant of the code in Figure 2.7, where the inputs and the output are passed as references. In both figures, the addresses of the input variables, `pub`, `key`, `mask`, are stored in registers `r0`, `r1`, and `r2`, respectively. To mark the value transitions in memory M and registers, e.g. $r0$, we denote all public values with `p`, including the addresses to the program input variables and the initial value in the memory bus. At line 3, the implementation loads the secret value from the address in register `r1` to register `r1`. These instructions lead to two leaks, one register-reuse transitional leakage in register `r1` and one memory-bus transition leakage (circled). That is, the initial value of `r1` is public (address to the value `key`), and the initial value in the memory bus is public, which we assume in this work. At line 4, the code loads the mask, which leads to no leaks. However, at line 5, we have a register-reuse leakage as in the previous example (Figure 3.2a) because the result of the exclusive-OR operation is stored at the same register as value `mask`, which leads to a leak related to the value of `key` (circled). To generate secure code, SecOpt needs to schedule the memory operations in a specific order ensure that there are no register-reuse and memory-bus leaks. To do that, SecOpt uses a stack slot to store the random value `mask` (line 4), then load the secret value (line 5), and finally load the random value again (line 6). One of these load operations may be optimized away, however, Unison does not allow the allocation of unused variables. Forcing the constraint model to consider dead copies is part of future work.

Constant-Resource Code

SecOpt aims at generating constant-resource code, where secret-dependent branches are balanced. Although in some cases it is possible to generate constant-resource code, in other cases, this is not possible because the source code is not balanced or because front- and middle-end compiler transformations have removed the balancing (dead) code. To enable mitigation of such code, we perform two program transformations, 1) add an empty basic block that the solver will fill with NOP operations and 2) add a basic block that consists of instructions from the basic block to balance that are deactivated (only in the case of one basic block).

Figure 3.4 demonstrates our transformations for a simple program that returns one if the input variables `key` and `pub` are equal and zero, otherwise. The first version of the transformed code (Figure 3.4a) adds an empty `else` block in the code, while the second version (Figure 3.4b) copies the assignment of `mask` to one in the `if` block to a newly defined `else` block.

To preserve the constant-resource property, the constraint model enforces all paths starting from a secret-dependent control-flow operation to have equal latency. Identifying these paths requires a prior analysis, which first identifies all secret-dependent control-flow instructions and, subsequently, finds all program paths that begin from these instructions. This analysis generates a set of lists of paths, where each list, $paths_{sec}$, consists of a set of paths that start from the same secret-

```
1 u32 check_bit(u32 pub, u32 key) {
2   u32 t = 0;
3   if (pub == key)
4     t = 1;
5   else
6     // nop;
7   return t;
8 }
```

(a) Add Empty Block

```
2   u32 _t, t = 0;
3   if (pub == key)
4     t = 1;
5   else
6     _t = 1;
7   return t;
8 }
```

(b) Copy Unbalanced Block

Figure 3.4: Balancing transformations

dependent control-flow instruction. The following constraint enforces the same execution time for each path in $paths_{sec}$.

`balance_blocks(`$paths_{sec}$`):`
$\forall p_1, p_2 \in paths_{sec}. \sum_{b \in p_1} \texttt{cost}(b) = \sum_{b \in p_2} \texttt{cost}(b)$

Figure 3.5a shows an implementation of the code snippet in Figure 3.4b in assembly code for processor ARM Cortex M0. First, the code (line 3) copies value zero to register `r2` (variable `t` in Figure 3.4b). At line 4, the implementation compares the two input variables, and if they are not equal (taken branch), the control flow goes to line 9; otherwise, it continues to line 7. These branches depend on the secret value in register `r1`, and hence, the attacker should not distinguish the execution time regardless of the branch destination. The not taken branch starting at line 7 copies value `#1` to the return register `r0` (1 cycle) and then branches to the exit block `.LBB0_3` (3 cycles). If the branch is taken, then there is an additional overhead of two cycles because the processor needs to calculate the target address and/or the comparison result. The taken branch starting at line 9 copies the content of variable `r2` to the return register (1 cycle) and assigns value `#1` to register `r1` (1 cycle), which corresponds to the unused temporary `_t` in Figure 3.4b. In total, each branch takes four cycles.

Composability of Security Mitigations

One of the major advantages of constraint-based approaches is composability of multiple properties in the form of constraints. The constraint-based approach of SecOpt allows the combination of multiple mitigations against different attacks. As we show in Publication 4, fine-grained software diversification may conflict with both constant-resource programming and software masking. SecOpt allows combining multiple mitigations while preserving the properties of these mitigations at the same time. This work focuses on fine-grained software diversification and software masking or constant-resource programming. The problem statement in this problem is the following:

```
1  @ r0: pub, r1: key
2  @ BB#0:
3    movs  r2, #0
4    cmp   r1, r0
5    bne   .LBB0_2
6  @ BB#1:
7    movs  r0, #1
8    b     .LBB0_3
9  .LBB0_2:
10   mov   r0, r2
11   movs  r1, #1
12 .LBB0_3:
13   bx    lr
14 ...
15 ...
```

(a) Secure variant 1

```
2  @ BB#0:
3    movs  r2, #0
4    cmp   r1, r0
5    bne   .LBB0_2
6  @ BB#1:
7    movs  r3, #1
8    mov   r0, r3
9    b     .LBB0_3
10 .LBB0_2:
11   mov   r3, r2
12   mov   r0, r3
13   movs  r3, #1
14 .LBB0_3:
15   bx    lr
```

(b) Secure variant 2

Figure 3.5: Two program implementations that preserve constant-resource property for ARM Cortex M0

Definition 3 *Secure Constraint-Based Code Diversification: Given problem* $P = \langle V, U, C, O \rangle$ *that describes a constraint-based compiler backend, we add constraints* C_{sec}*, that protect against side-channel vulnerabilities,* $P_{sec} = \langle V, U, C \cup C_{sec}, O \rangle$*. We aim at generating a set,* S*, of solutions to the constraint problem,* P_{sec} *that are different with each other,* $\forall p_i, p_j \in S.\delta(p_i, p_j)$ *and are functionally equivalent* $\forall p_i, p_j \in S.p_i \sim p_j$*.*

To achieve these goals, we combine the constraints and methods we discussed in the previous sections. In particular, the combined approach extends the set of constraints with constraints `balance_blocks` or `conflict_rassign` and `conflict_order` and then, uses LNS to find diverse program solutions. Figure 3.5 shows two program variants that preserve the constant-resource property based on the copy transformation in Figure 3.4b. The two variants balance the execution time of the two paths that split at the branch instruction `bne`. In both implementations, the first path consists of basic block one, `BB#1`, and the second consists of basic block two, `.LBB0_2`. The two variants differ with regard to the register assignment, including copying a result from one register to another (lines 7-8 and 11-12). The two variants differ in the code size, which may result in different addresses in the binary code. This relocation, together with different register assignment, e.g. `movs r1, #1` (line 11) and `movs r3, #1` (line 13), respectively, alter the semantics of possible gadgets and may break code-reuse gadgets that consider either of the variants. Performing code re-randomization may further harden the implementation against both attacks.

Verification of Security Properties

Code verification on binary code tests security properties in the binary code to verify the preservation of these security properties or identify security vulnerabilities. Typically, code verification approaches rely on formal methods to provide guarantees that the requested security properties hold. There are different code-verification methods, including symbolic execution that uses constraint solving to prove the requested properties and abstract interpretation that depends on a formal abstraction of the program values and semantics.

SecOpt generates secure code against timing side channels. After code generation, we verify the constant-resource property in the generated code to increase the trust in SecOpt. In particular, to ensure that the constant-resource property holds in the generated code, we verify this property using external static-analysis tools. More specifically, for each processor we target, we use a Worst Case Execution Time (WCET) tool to verify that secret-dependent paths take the same time. WCET analysis generates an overapproximation (and optionally an underapproximation, Best Case Execution Time (BCET)) of the execution time of a program. WCET analysis is an important topic in embedded systems because it is an input to schedulability analysis, which tests whether the system under analysis meets its deadlines. To verify the constant-resource property, we derive the WCET and the BCET using symbolic values for the secret input variables of the function under analysis and compare these values. If the two values are equal, we have an indication that the execution time does not depend on these secret values. If the two values are not equal, then we need to investigate further if the execution time depends on secret values, or if the difference between WCET and BCET is due to overapproximation of the analysis.

In some cases, secure compilation uses verification approaches to accept compiler-generated code if the verification test succeeds or reject the code if the process identifies security vulnerabilities [62]. In this work, we verify the constant-time property in WebAssembly. Our approach uses relational symbolic execution, a method that performs symbolic execution on a program with two input states or two programs with the same input state [60]. For testing the constant-time property, we execute symbolically two executions of the same program with two input states. The two states differ with regard to the secret input values, which take different initial symbolic values. Then, the analysis executes the program symbolically using one state that considers both executions. The analysis is possible by using paired variables, each consisting of two versions that correspond to the different executions. Given that the two states differ only with regard to the secret values, a potential execution path divergence signifies a timing leakage. Similarly, the analysis needs to ensure that there are no secret-dependent memory operations, which allow cache timing attacks. More formally, we consider $\{\Phi\}\langle(M_1, c)|(M_2, c)\rangle\{\Psi\}$, where Φ is the precondition, namely the security policy that defines which input variables are secret or public. Ψ denotes the verification properties, namely the absence of secret-dependent control-flow instructions and

secret-dependent memory operations. M_1 and M_2 are the initial memory instances, and c is the program instructions. When discovering a vulnerability, the approach returns the vulnerability as a solution to the requirement.

Loop analysis is a challenge in symbolic execution because it may lead to path explosion. Among different methods to deal with loops in symbolic execution are using bounded loop unrolling or loop invariants. The former may lead to increased analysis overhead and unsound analysis, whereas the latter is challenging to perform automatically. Our work considers both unbounded loop unrolling and the generation of a relational invariant.

3.2 Methodology

The topic of this dissertation combines cybersecurity and computer science methods, which include theoretical research and applied experimental research [63].

In the bibliography, there are several compiler-based approaches to tackle security properties. The process of conducting research for each research question starts with a survey in the area to identify challenges in state-of-the-art research. We identify the need for highly optimizing security approaches and evaluating the effect of the sequential composition of multiple mitigations. After defining the problem, we follow an iterative process that consists of three steps, 1) define a hypothesis, 2) implement/extend a research prototype to evaluate the hypothesis, 3) refine the hypothesis based on the results. This process repeats until the final research project is complete. The work towards this dissertation consists of the following incremental steps:

- Design and evaluate an LNS-based algorithm as an extension to Unison to generate highly diverse solutions in Mips. This step is the first part of the development of SecOpt.
- Design and evaluate a new LNS-based algorithm based on the structural decomposition of a function. Evaluate this approach in a whole-program diversification scheme for Mips32.
- Extend SecOpt to optimize code that preserves software masking and supports the ARM Cortex M0 processor. As part of this step, we prove that the proposed constraint model leads to code that does not leak secret information through transitional leakages due to register reuse and memory-bus reuse.
- Extend SecOpt to optimize code that preserves the constant-resource property and combine with diversification.

Apart from these incremental steps towards SecOpt, we investigate a different approach to perform whole-program verification of the constant-time property in WebAssembly.

We evaluate Publications 1 and 2 using benchmark functions from two benchmark suites, MediaBench [64] and SPEC CPU2006 [65] that are popular for evaluating embedded-system and compiler-based approaches. Publication 1 uses 17 small, randomly selected functions from both benchmark suites. The evaluation shows that the proposed LNS-based approach trades scalability for diversity in CP. In addition, the evaluation shows our approach allows the generation of multiple optimal diverse solutions for the majority of the benchmarks. Relaxing the optimality constraint enables more diverse solutions. Publication 2 uses 20 medium-size functions from MediaBench. The evaluation of Publication 2 confirms the results of Publication 1 for larger programs. In addition, Publication 2 presents a whole-program diversification evaluation using a case study from MediaBench, G.721. This application is an implementation of a set of voice compression algorithms. We use a GCC-based tool to link the generated variants for the MIPS32-based Pic32MX microcontroller. This case study shows up to 95% code-reuse gadget diversification or relocation.

Publications 3, 4, and 5 use cryptographic implementations from the real world and previous work that evaluates similar approaches. The evaluation of Publication 3 uses real-world constant-time implementations from diverse libraries, including libsodium [66], HACL* [67], BearSSL [68]. In addition, the evaluation uses known timing vulnerabilities from the literature. This publication presents two approaches to verify constant-time programs. The first approach uses unbound loop unrolling and is able to verify 55 out of 57 implementations. The second approach uses a lightweight relational invariant generation and is able to verify the rest two implementations but fails to analyze many implementations due to the limited precision of the generated invariant.

Publication 4 applies theoretical and experimental research methodology in the evaluation. To ensure security, we prove that the security constraint model implies secure code generation based on our leakage model. In addition, the evaluation in Publication 4 uses twelve benchmark functions that we derive from previous work [25] to perform an empirical evaluation. We evaluate the performance overhead of the generated code and the compilation-time overhead compared with Unison. To evaluate the speedup of our approach compared to other security-aware approaches, we compare with the approach by Wang et al. [25] and LLVM with no optimizations, -O0. This evaluation shows a high speedup of up to three times faster than LLVM -O0 and the work by Wang et al. [25] at the expense of a compilation time increase.

The evaluation of Publication 5 uses the same benchmarks as Publication 4 and an additional set of five benchmarks to evaluate the constant-resource property. These benchmark functions comprise diverse algorithms that may take secret values as inputs and contain secret-dependent control-flow instructions [69]. To verify the security of the generated solutions, we use two WCET tools for ARM Cortex M0 [70] and Mips [71, 72]. These tools calculate the worst- and best-case execution time, and we use them to show that the generated variants do not depend on secret values. The evaluation indicates that property-preserving diversification introduces

diversification-time overhead, however, this does not reduce the effectiveness against code-reuse attacks.

Chapter 4

Related Work

This chapter presents state-of-the-art research in defenses against code-reuse attacks (Section 4.1), and side-channel attacks (Section 4.2). The latter consists of three parts and includes related work on mitigations against timing side-channel attacks, mitigations against power side-channel attacks, and verification approaches for timing side-channel attacks.

4.1 Code-Reuse Attacks Mitigations

Table 4.1: Mitigation approaches against code-reuse attacks

Pub.	Mitigation	InL	OutL	ML	Av.
Abadi et al. [35]	CFI	x86	x86	Bin	✗
Pappas et al. [8]	Div	x86	x86	Bin	✓
Homescu et al. [9]	Div	C, C++	x86	llvm	✓
AVRAND [73]	Div, RR	AVR	AVR	Bin	✓
C-Flat [74]	CFI	ARM	ARM	Bin	✓
CFI CaRE [75]	CFI	ARM	ARM	Bin	✗
Koo et al. [76]	Div, RR	C, C++	x86	llvm	✓
MicroGuard [1]	Div, CFI	C, C++	ARM	llvm/Bin	✗
HARM [77]	Div, RR	ARM	ARM	Bin	✓
FH-CFI [78]	HWCFI	ARM	ARM	Bin	✗
SecOpt [17, 18, 21]	Div	C, C++	Mips, ARM	llvm	✓

Code-reuse attacks constitute a serious threat to computer software in both high-end computers [29] and embedded systems [27, 28]. There are two main mechanisms to mitigate code-reuse attacks, CFI [34] and automatic software diversification [7]. CFI includes hardware and software mechanisms to prevent illegitimate

control-flow violations during program execution. On the other end, automatic software diversification hinders code-reuse attacks by introducing uncertainty to the program implementation. This uncertainty affects the location and exact implementation of code-reuse gadgets, which are the building blocks of code-reuse attacks.

Table 4.1 shows a number of representative approaches against code-reuse attacks. The table includes the publication citation (Pub.), the mitigation each approach uses (Mitigation), the input language (InL), the output language (OutL), the mitigation level (ML), which is either at binary code (Bin) or a compiler (e.g. llvm), and finally, the availability of the respective artifact (Av.). For the availability field (Av.), ✓indicates that the artifact is available, whereas ✗indicates that the main author of this dissertation was not able to find the artifact.

Table 4.1 shows a set of approaches that use CFI against code-reuse attacks [35, 74, 75, 1, 78]. The majority of these approaches target embedded systems, including C-Flat [74], CFI CaRE [75], and FH-CFI [78], and MicroGuard [1] that target ARM systems. One of these approaches, MicroGuard [1] combines CFI with software diversification against code-reuse attacks. In general, CFI approaches lead to higher execution-time overhead than software diversification approaches [34, 7].

Automatic software diversification (Div in Table 4.1) is another approach against code-reuse attacks. ASLR is a coarse-grained software diversification method that randomly selects the address space of key data areas, such as the address of dynamic libraries. ASLR is the most widely-used diversification method, and its effect on performance is insignificant. However, ASLR leads to low entropy, which enables brute-force code-reuse attacks [79]. Fine-grained software diversification, which includes diversification at the function- or instruction-level of the program, provides improved protection against code-reuse attacks.

Pappas et al. [8] perform fine-grained software diversification at the binary level and apply zero-cost transformations, namely register randomization, instruction schedule randomization, and function shuffling. However, they do not evaluate the actual performance overhead of their approach. In contrast, SecOpt uses a cost model to calculate the performance overhead and allows diversification with no performance degradation. Also, SecOpt enables more transformations including NOP insertion, register copying, spilling, and constant rematerialization. SecOpt may be combined with function shuffling to achieve whole-program diversification [18].

Homescu et al. [9] present a fine-grained diversification approach that inserts NOP instructions to the code. To reduce the introduced overhead, they use profiling information to prioritize NOP insertion in pieces of code that have low execution frequency. Seibert et al. [33] show that static frequency NOP insertion is possible to bypass using side-channel information. SecOpt is also able to control the introduced overhead by using a static cost model, while it allows targeted diversification in code without introducing performance overhead. The latter is possible because SecOpt uses a larger variety of transformations than NOP insertion.

The introduction of advanced code-reuse attacks allows for deciphering the diversification scheme by using a memory vulnerability to read the program mem-

ory [31, 32], or using timing information [33]. These attacks give rise to re-randomization (RR in Table 4.1) approaches. Re-randomization typically switches between different program variants at specific time intervals [76] or at different events, such as at reboot time [73]. Re-randomization may introduce additional runtime performance overhead that may be low, such as HARM [77] that introduces 5% overhead. SecOpt may be used in a re-randomization scheme at boot time against attacks, such as BROP [32]. We leave the evaluation of such an approach as future work.

4.2 Defending Side-Channel Attacks

Side-Channel attacks constitute a serious threat to cryptographic implementations. There are different side-channel attacks, including timing, power, electromagnetic, and sound side-channel attacks. This dissertation focuses on timing and power side-channel mitigations. The next sections present a set of representative mitigation approaches against these attacks.

Defending Timing Side-Channel Attacks

In this section, we discuss methods to mitigate and verify timing side-channel attacks.

Timing Side-Channel Mitigations

Table 4.2: Mitigation approaches against timing side-channel attacks

Pub.	Attack	Mitig.	InL	OutL	ML	Av.
Crane et al. [14]	TSC	Div	C, C++	x86[a]	llvm	✗
Raccoon [15]	TSC	Obf	C, C++	x86	llvm	✗
Fact [80]	TSC	CT	DSL	C	Custom	✓
HACL* [81]	TSC, MC	CT	DSL	C	F_{low}	✓
Jasmin [82]	TSC, MC	CT	DSL	x86	Custom	✓
Winderix et al. [61]	TSC, IL	BB	C, C++	MSP430	llvm	✓
Constantine [83]	TSC	CT	C, C++	x86	llvm	✓
Crow [84]	TSC	Div	C, C++	Wasm	llvm	✓
Vu et al. [4]	VBL, TSC	SM,CT	C, C++	ASM	llvm	✗
SecOpt [21]	TSC, TBL, CRA	Div, SM/BB	C, C++	Mips, ARM	llvm	✓

[a]The evaluation targets x86, however the method applies to other architectures

There are different approaches to mitigate timing side-channel attacks. These approaches either eliminate the secret-dependent timing differences in the program execution [85] or obfuscate the timing profile of the program to reduce the ability of the attacker to identify secret values [15, 14] Table 4.2 shows a set of representative mitigation approaches against timing side-channel attacks (TSC). The table includes the publication (Pub.), the attack, the approach mitigates (Attack), the mitigation the approach applies (Mitig.), the input language (InL), the output language (OutL), the mitigation level (ML), which is a custom compiler (custom), a specific compiler (F_{low}), or the LLVM compiler (llvm). The last field (Av.) indicates that the artifact is available (✓) or not available (✗).

There are two main approaches that eliminate secret-dependent time differences, 1) cryptographic constant-time (CT) programming discipline [85], which eliminates secret-dependent timing differences by rewriting the code, and 2) constant-resource (BB) programming [45], which instead balances the different execution paths to take the same time.

The constant-time programming discipline replaces secret-dependent branch and memory operations with constant-time equivalent that make use of logic operations. HACL* [81] is an approach to generate code that is constant time and memory safe against memory corruption (MC). The output code is in C, and thus, there is another compilation step from C to assembly code that may use CompCert [86], a verified compiler. Jasmin [82] is low-level optimizing cryptographic Domain Specific Languages (DSLs) that generate efficient constant-time code for cryptographic implementations. The main drawback of Jasmin is that the input code is written in a low-level language, which requires re-implementing legacy cryptographic algorithm implementations and acquiring a deep understanding of low-level code.

Constant-resource programming is a more relaxed mitigation approach compared to the constant-time programming discipline. In particular, constant-resource programming does not require the absence of secret-dependent branches. Instead, it allows balancing secret-dependent branches with NOP instructions to hinder the attacker from identifying the selected execution path. Winderix et al. [61] implement an approach to balance secret-dependent branches on MSP430. Their approach protects against both timing attacks and interrupt-latency (IL) side-channel attacks. SecOpt focuses on timing attacks and achieves balancing secret-dependent branches with up to 70% overhead.

Constantine [83] is a different approach that achieves constant time by automatically linearizing code. Their approach introduces large overhead of up to five times. Raccoon [15] uses obfuscation (Obf) to hide secret-dependent leaks. The main disadvantage of this approach is high performance overhead of up to 16 times. Crane et al. [14] mitigate timing side channels using fine-grained code diversification (Div) by inserting memory NOP operations. However, their approach may lead to up to 8 times performance overhead. The main disadvantage of these approaches is high execution-time overhead compared to SecOpt that introduces up to 70% overhead.

Verification

Code verification is a way to identify timing vulnerabilities in programs. Almeida et al. [85] use product programs to verify constant-time programs in C. Vale [87] verifies the correctness, safety, and security of binary code in ARM and x86. Among other security properties, Vale preserves the constant-time property using taint analysis. Binsec/Rel [24] performs relational symbolic execution [60] to verify constant-time program in binary code.

The verification approach of this thesis, Vivienne [19] uses also relational symbolic execution to verify the constant-time property in WebAssembly code. In addition, Vivienne implements a lightweight invariant inference approach. Bastys et al. [88] is another approach that uses concolic execution to verify the constant-time property in WebAssembly.

Power Side-Channel Mitigations

Table 4.3: Mitigation approaches against power side-channel attacks

Pub.	Attack	Mitig.	InL	OutL	ML	Av.
Eldib and Wang [89]	VBL	SM	DSL	-	Custom	✗
Papagiannopoulos and Veshchikov [13]	TBL	SM	AVR	AVR	Binary	✓
Besson et al. [62]	IFL	-	C	ASM	CompCert	✗
Wang et al. [25]	TBL	SM	C, C++	ASM	llvm	✓
Athanasiou et al. [26]	TBL	SM	ARM	ARM	Binary	✓✗
Vu et al. [4]	VBL, TSC	SM,CT	C, C++	ASM	llvm	✗
Rosita [47]	TBL	SM	ARM	ARM	Binary	✓
SecOpt [20]	TBL	SM	C, C++	Mips, ARM	llvm	✓

Power side-channel attacks record the power traces of a computer to extract secret values, such as cryptographic keys. There are different mitigation approaches to hinder power side channel attacks during the program execution. An approach to mitigate power side channels is to randomize the secret data, so that the power traces do not reveal secret information to the attacker. Software masking (SM), uses the exclusive-OR operation, $\oplus$, to mix the secret value with a randomly generated value. This randomly generated value, or mask, allows the randomization of the secret value and requires the same mask to decipher.

We consider different types of power leakage, Value-Based Leakage (VBL) and Transition-Based Leakage (TBL). VBLs appear when secret values are not masked,

i.e. public values known to the attacker interact with secret values [89, 4], whereas, TBL appear when fundamental hardware structures, such as hardware registers, memory cells, and memory bus, leak information by transitioning from one value to another. The absence of VBLs does not guarantee the absence of TBLs, whereas the opposite is true.

Table 4.3 shows a set of representative mitigation approaches against power side-channel attacks. For each of the approaches, Table 4.3 shows the publication (Pub.), the attack the approach mitigates (Attack), the mitigation the approach applies (Mitig.), the input language (InL), the output language (OutL), the mitigation level (ML), which is either binary or a compiler, like CompCert or LLVM (llvm). The last field (Av.) indicates that the artifact is available (✓), not available (✗), parts of the artifact are missing (✓✗).

The approaches by Papagiannopoulos and Veshchikov [13] and Rosita [47] are processor specific, focusing on AVR and ARM Cortex M0, respectively. They mitigate different types of TBLs, including register-reuse leakage, memory-reuse leakage, and memory-bus-reuse leakage, which are detected in a specific processor implementations.

Wang et al. [25] take as input masked code and use a more generic type-inference-based approach [90] to identify possible register-reuse leaks and subsequently mitigate them. The main disadvantage of this approach is that it does not generate highly optimized code. SecOpt follows a similar type-inference-based approach to optimize masked code with no register-reuse leaks and memory-bus reuse leaks.

Athanasiou et al. [26] use the same type-inference approach to find and mitigate possible register-reuse leakages. SecOpt generates code that is free from register-reuse leaks and memory-bus reuse leaks.

4.3 Secure Compilation and Optimization

Popular languages like C enable security vulnerabilities, such as memory corruption and many undefined behaviors [3]. Compiler development and research focus mostly on functional correctness in accordance with the language specification. In addition, general-purpose compilers focus on optimizing the performance efficiency or the size of the code, however, they rarely consider security properties [3]. Therefore, important compiler algorithms and heuristics are designed with performance and code size in mind and not security. With the advent and popularity of the Internet and recently the IoT devices, where multiple computers connect to each other, security has become a major concern. Secure compilation is a field that aims at generating secure code, where performance is a secondary aspect.

Table 4.3 presents approaches that combine security features with highly optimized code. Jasmin [82] is an approach that generates secure and optimized code. The main disadvantage of Jasmin is that it uses a low-level DSL, that requires writing cryptographic code in a new assembly-like language.

Vu et al. [91] present an approach that prevents compiler-introduced vulnerabilities in LLVM. Vu et al. [91] generate highly optimized code that preserves security properties, such as software masking (SM) against VBLs and the constant-time property for timing side channels (TSC). Unfortunately, the artifact for their approach is not available.

A different approach by Besson et al. [62] proves that the compiler preserves security properties. In particular, they show that two optimization passes in CompCert [86] preserve information-flow properties at function entries and exits.

Chapter 5

Summary of Publications

The following sections (Sections 5.1 to 5.5), provide a summary of each paper that is included in this dissertation

5.1 Publication 1: Constraint-Based Software Diversification for Efficient Mitigation of Code-Reuse Attacks

Fine-grained software diversification is an approach that is effective against code-reuse attacks. Related work focuses on high-end computer architectures, with little focus on embedded devices, although, code-reuse attacks target embedded devices [28, 29, 30]. An additional advantage of fine-grained software diversification is its low introduced overhead, which makes it suitable for resource-constrained devices. Unfortunately, many related approaches do not control the introduced performance overhead.

This publication presents a fine-grained software diversification approach that uses CP to enable control over the introduced performance overhead. A wide range of low-level program transformations in the underlying compiler backend enables fine-grained diversification that targets Mips32, an embedded-system architecture. The underlying constraint-based compiler backend generates optimized code with regard to performance and code size. To achieve high diversity and scalability, this publication proposes LNS, a local-search-based heuristic, which attempts to find solutions to the constraint model that correspond to machine-code implementations. The evaluation of the algorithm on 17 small functions from MediaBench and SPEC CPU 2006 shows that the presented algorithm enables the generation of program variants that are different from each other with an acceptable diversification time. The available diversification transformations allow the generation of zero-overhead variants, which corresponds to highly optimized function variants.

5.2 Publication 2: Constraint-Based Diversification of JOP Gadgets

Publication 1 presents an algorithm that is efficient and effective for small functions that represent around 24% of the total set of functions in MediaBench. This publication presents a different algorithm that allows the diversification of larger functions using structural decomposition. This algorithm first solves a subproblem that consists of the inter-block program-variable assignments and then, for each basic block, it generates multiple solutions that can be combined to generate diversified program variants. This publication also presents a distance measure that is adjusted to the code-reuse attack properties. The evaluation uses 20 functions from MediaBench that cover 96% of the function size in the bench suite. The evaluation shows that the global LNS-based algorithm is more effective against code-reuse attacks but scales up to around 60% of the functions in MediaBench, whereas the decomposition-based algorithm scales to up to 93% of the functions. This publication evaluates also the effect of whole-program diversification on a case study. This case study shows that the proposed diversification algorithms have high effectiveness against code-reuse attacks while performing additional randomization steps, such as function shuffling, improves the effectiveness of diversification measured by gadget relocation.

5.3 Publication 3: Vivienne: Relational Verification of Cryptographic Implementations in WebAssembly

Timing side-channel attacks constitute a serious threat to the security of cryptographic implementations. Most languages and many algorithm implementations are vulnerable to side-channel attacks, including WebAssembly, a new language for the web, which is portable, efficient, and features security properties. The characteristics of WebAssembly make it a suitable choice for implementing cryptographic libraries with multiple cryptographic implementations already available.

Relational verification is an efficient approach for verifying programs that follow the constant-time policy and/or locate constant-time violations. Vivienne presents an approach that uses relational symbolic execution to identify timing vulnerabilities in WebAssembly. Loop analysis is the main bottleneck of symbolic execution. To deal with this, this publication proposes an approach to automatic relational invariant generation.

The evaluation uses 57 cryptographic library implementations and shows that relational symbolic execution with loop unrolling is efficient for the verification of the constant-time property, when the loop bounds of the analyzed program are not very high. For the benchmarks that contain large loop bounds, relational invariant generation is more effective than unbound loop unrolling. However, sometimes the automatic relational invariant does not capture the loop bounds, which results in low effectiveness in analyzing compiler-generated code.

5.4 Publication 4: Securing Optimized Code Against Power Side Channels

Power side channels are a serious threat to cryptographic libraries. Power attacks typically require access to the victim's physical location and record the power consumption of the target machine in time using devices, such as an oscilloscope. These attacks are very powerful because they can identify small variations in the power consumption of the executing program.

Software masking is a software mitigation against power side-channel attacks, which hides secret values from the power traces using randomly generated variables. These random values statistically remove the dependencies of the secret values from the power traces.

Secret-dependent transitional effects, for example, when hardware registers or the memory bus transition from one value to another, may leak secret information through power side channels. Although the source code of a program may be masked correctly, the compiler may invalidate these transformations by, for example, reusing hardware registers. This publication presents a compiler-based approach that generates highly optimized code that mitigates power side-channel attacks. The paper presents a formal proof that the proposed constraint model is correct with regard to the leakage model. The evaluation of the approach on twelve masked programs targeting two embedded architectures, ARM Cortex M0 and Mips, shows that our approach leads up to 13% performance overhead compared to optimal non-secure compilation and a geometric mean speedup of approximately three compared to other secure-compilation approaches.

5.5 Publication 5: Thwarting Code-Reuse and Side-Channel Attacks in Embedded Systems

Security protection of a computing system typically requires application of multiple mitigations against different attacks that may affect the system. Applying these mitigations sequentially may lead to mitigation conflicts, when the latest applied mitigation reverts or invalidates the changes of the previously applied mitigations. Embedded systems have additional constraints that derive from resource limitations, such as battery life, and thus, overall resource estimation is of major importance.

Publication 5 concerns thwarting code-reuse attacks and side-channel attacks in embedded systems. An efficient mitigation against code-reuse attacks is fine-grained code diversification, while typical mitigations against power and timing side-channel attacks include low-level code transformations. Both fine-grained code diversification and side-channel mitigations operate at the low-level implementation of the input code. The evaluation runs on 15 benchmark programs derived from previous work against side-channel attacks. This publication shows that 1) fine-grained software diversification may break side-channel mitigations, 2) a combined

mitigation against both code-reuse attacks and side-channel attack is feasible but with increased compilation overhead, and 3) there is no clear negative effect of the effectiveness of diversification against code-reuse attacks, when protecting also against side-channel attacks.

Chapter 6

Conclusion and Future Work

This chapter presents the conclusion of this dissertation (Section 6.1) and discusses future research directions (Section 6.2).

6.1 Summary of Contributions

This dissertation presents a combinatorial approach to secure code generation. This work features three properties: performance awareness, composability, and formalisation. Below we summarise the answers to the research questions of this dissertation and discuss the obtained research results.

RQ1: How feasible and effective is performance-aware constrained-based software diversification against code-reuse attacks?

Our experiments show that performance-aware constraint-based software diversification can effectively diversify code-reuse gadgets. The proposed algorithm allows for the efficient generation of highly diverse solutions, while it controls the generated performance overhead. Our approach scales to up to medium-sized programs of approximately 500 Machine Intermediate Representation (MIR) instructions. Hence, we believe that our approach constitutes an important building block for whole-program diversification or re-randomization to provide high effectiveness against code-reuse attacks.

RQ2: How feasible is secure constraint-based optimization of cryptographic implementations?

Our approach to constraint-based optimization of cryptographic implementation is highly optimizing for small cryptographic functions of up to 100 MIR instructions. The results that generate programs free of transitional leaks show a high speedup

compared to competing approaches and non-optimized code. This is achieved at the expense of compilation time. Constant-resource preservation is effective against timing side channels and generates secure low-level code.

RQ3: How feasible and effective is a combined mitigation against code-reuse attacks and side-channel attacks?

Our experiments show that combining software diversification with software mitigations against side-channel attacks enables the generation of multiple program variants that are secure against side-channel attacks. There is an overhead on the diversification time, however, there is no clear effect on the effectiveness of diversifying code-reuse gadgets and thus hindering code-reuse attacks.

RQ4: How feasible is code verification of binary code against timing side channels?

Our constant-resource verification approach verifies the constant-resource property in all generated programs by SecOpt. Similarly, our approach to relation verification to test the constant-time property in WebAssembly is able to analyze successfully 55 out of 57 real-world implementations consisting of large code bases. In addition, the relational invariant approach is able to analyze the remaining two implementations, while a more precise invariant generation approach may successfully analyze the total set of implementations.

Conclusion

This dissertation developed a constraint-based compiler backend approach that presents a concrete step towards secure-by-design optimized compilation. The main features of this approach are composability of security measures, performance awareness that allows the generation of highly optimized code, and a constraint-based framework that exhibits properties that have been formalised. Code verification is an additional step toward highly trusted code generation. To summarize, this work proposed a novel compiler-based approach to generate highly optimized and secure code against major vulnerabilities that affect security-critical software.

6.2 Future Work

There are several directions for future work that can focus on improvements and extensions to proposed approaches. The extension of the current work could include 1) evaluating our approach against full-fledged attacks, 2) extending the software-masking optimization approach to larger programs by decomposing linearized programs into smaller pieces, and 3) extending SecOpt to support more cybersecurity mitigations. Below we discuss the directions of the future work in detail.

Evaluation against Complete Attacks

This dissertation concerns the automatic and correct-by-design generation of code that satisfies security mitigations. Our work evaluates these approaches using formal methods (see Publication 4), empirical evaluation (see Publication 3 and 5), or statistical properties (see Publication 1 and 2). However, an interesting research direction would be to investigate the effect on full-fledged attacks. Two examples of attacks include code-reuse attacks and power side-channel attacks that have recently received increased interest.

Software diversification provides statistical properties that hinder code-reuse attacks. However, an interesting direction is to design different types of code-reuse attacks, such as ROP and JOP attacks, in Mips and ARM. This research direction may give further insights into how to improve and target diversification towards full-fledged attacks.

Power side-channel attacks have advanced significantly in the recent years, including statistical analysis of the power traces using deep learning. These attacks are powerful and evaluating our approach against such attacks may provide additional insights on extending the mitigation of our approach to further transitional leaks.

Scalability Enhancement

Scalability, namely the ability to analyze large problems is an active and demanding research topic in combinatorial optimization. This dissertation has made clear steps toward increased scalability in Publication 2. In addition, Publication 3 investigates additional solving methods to enhance the scalability of the optimization approach. The introduction of security constraints increases the complexity of the problem and, hence, its compilation time. However, Publication 3 uses linearized input programs that consist of a single basic block. One step towards improving the scalability of the approach in Publication 3 is extending the security analysis to support if statements and loops. This direction may reduce the accuracy of the security analysis and lead to reduced code efficiency.

Cybersecurity Countermeasures

This dissertation concerns mitigations against code-reuse attacks and side-channel attacks. However, there are additional attacks that depend on compiler-generated code.

Memory-Probing Attacks

Memory-probing attacks allow an adversary to read the content of the main memory and/or the register file [92]. One idea is to use SecOpt to reduce the presence of secret values in memory by reducing the live range of registers and stack operations. Overwriting the secret data in memory, including the stack and register after the end

of the live range is an additional transformation towards reducing the capabilities of memory-probing attacks.

Speculation Attacks

Modern processors use speculation to improve the performance of the program execution when the result of a branch is not known. In particular, the processor uses statistical information from previous branches to take a branching decision before the processor calculates the actual branch decision. Speculative execution improves the processor performance when the branch prediction is correct. In case of misprediction, the processor discards the speculatively executed instruction results. However, the processor does not reverse any side effects of the speculative execution, such as cache updates. A compiler-based approach to mitigate these attacks may include padding vulnerable branches with NOPs that delay the speculative execution of instructions that leak secret information. This approach may introduce high performance overhead. Instead, verification of the absence of speculation leaks using relational verification is an effective method to ensure the security of the generated software [93].

References

[1] M. Salehi, D. Hughes, and B. Crispo, "MicroGuard: Securing Bare-Metal Microcontrollers against Code-Reuse Attacks," in *2019 IEEE Conference on Dependable and Secure Computing (DSC)*, Nov. 2019, pp. 1–8.

[2] A. Bendovschi, "Cyber-Attacks – Trends, Patterns and Security Countermeasures," *Procedia Economics and Finance*, vol. 28, pp. 24–31, Jan. 2015.

[3] V. D'Silva, M. Payer, and D. Song, "The Correctness-Security Gap in Compiler Optimization," in *2015 IEEE Secur. Priv. Workshop*, 2015, pp. 73–87.

[4] S. T. Vu, A. Cohen, A. De Grandmaison, C. Guillon, and K. Heydemann, "Reconciling optimization with secure compilation," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–30, 2021.

[5] R. C. Lozano, M. Carlsson, G. H. Blindell, and C. Schulte, "Combinatorial Register Allocation and Instruction Scheduling," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 41, no. 3, pp. 17:1–17:53, 2019.

[6] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 552–561.

[7] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated Software Diversity," in *2014 IEEE Symposium on Security and Privacy*, May 2014, pp. 276–291.

[8] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization," in *2012 IEEE Symposium on Security and Privacy*, May 2012, pp. 601–615, iSSN: 1081-6011.

[9] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided Automated Software Diversity," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, ser. CGO '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1–11.

[10] P. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *Advances in Cryptology — CRYPTO' 99*, ser. Lecture Notes in Computer Science. Springer, 1999, pp. 388–397.

[11] E. Brier, C. Clavier, and F. Olivier, "Correlation Power Analysis with a Leakage Model," in *Cryptographic Hardware and Embedded Systems - CHES 2004*, ser. Lecture Notes in Computer Science. Springer, 2004, pp. 16–29.

[12] P. C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *Annual International Cryptology Conference*. Springer, 1996, pp. 104–113.

[13] K. Papagiannopoulos and N. Veshchikov, "Mind the Gap: Towards Secure 1st-Order Masking in Software," in *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 2017, pp. 282–297.

[14] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity," in *Proceedings 2015 Network and Distributed System Security Symposium*. Internet Society, 2015.

[15] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing Digital {Side-Channels} through Obfuscated Execution," in *26th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 431–446.

[16] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with WebAssembly," in *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, 2017, pp. 185–200.

[17] R. M. Tsoupidi, R. Castañeda Lozano, and B. Baudry, "Constraint-Based Software Diversification for Efficient Mitigation of Code-Reuse Attacks," in *International Conference on Principles and Practice of Constraint Programming*, 2020, pp. 791–808.

[18] R. M. Tsoupidi, R. Castañeda Lozano, and B. Baudry, "Constraint-based diversification of JOP gadgets," *Journal of Artificial Intelligence Research*, vol. 72, pp. 1471–1505, 2021.

[19] R. M. Tsoupidi, M. Balliu, and B. Baudry, "Vivienne: Relational Verification of Cryptographic Implementations in WebAssembly," in *2021 IEEE Secure Development Conference (SecDev)*, 2021, pp. 94–102.

[20] R. M. Tsoupidi, R. Castañeda Lozano, E. Troubitsyna, and P. Papadimitratos, "Securing Optimized Code Against Power Side Channels," in *2023 IEEE Security Foundations Symposium (CSF)*, 2023, to appear.

[21] R. M. Tsoupidi, E. Troubitsyna, and P. Papadimitratos, "Thwarting code-reuse and side-channel attacks in embedded systems," *arXiv preprint arXiv:2304.13458*, 2023.

[22] "ENISA Threat Landscape 2022." [Online]. Available: https://www.enisa.europa.eu/publications/enisa-threat-landscape-2022

[23] S. Liu and B. Cheng, "Cyberattacks: Why, what, who, and how," *IT Professional*, vol. 11, no. 3, pp. 14–21, 2009.

[24] L.-A. Daniel, S. Bardin, and T. Rezk, "Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1021–1038.

[25] J. Wang, C. Sung, and C. Wang, "Mitigating power side channels during compilation," in *Proc. 2019 27th ACM Jt. Meet. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, ser. ESEC/FSE 2019. Association for Computing Machinery, 2019, pp. 590–601.

[26] K. Athanasiou, T. Wahl, A. A. Ding, and Y. Fei, "Automatic Detection and Repair of Transition- Based Leakage in Software Binaries," in *Softw. Verification*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2020, pp. 50–67.

[27] G.-A. Jaloyan, K. Markantonakis, R. N. Akram, D. Robin, K. Mayes, and D. Naccache, "Return-Oriented Programming on RISC-V," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '20, Oct. 2020, pp. 471–480.

[28] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented Programming: A New Class of Code-reuse Attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '11. New York, NY, USA: ACM, 2011, pp. 30–40.

[29] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented Programming Without Returns," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10. New York, NY, USA: ACM, 2010, pp. 559–572.

[30] O. Gilles, F. Viguier, N. Kosmatov, and D. G. Pérez, "Control-flow integrity at risc: Attacking risc-v by jump-oriented programming," 2022.

[31] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi, "Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization," in *2013 IEEE Symposium on Security and Privacy*, May 2013, pp. 574–588.

[32] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking Blind," in *2014 IEEE Symposium on Security and Privacy*, May 2014, pp. 227–242, iSSN: 2375-1207.

[33] J. Seibert, H. Okhravi, and E. Söderström, "Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14, Nov. 2014, pp. 54–65.

[34] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-Flow Integrity: Precision, Security, and Performance," *ACM Computing Surveys*, vol. 50, no. 1, pp. 16:1–16:33, Apr. 2017.

[35] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*, ser. CCS '05, Nov. 2005, pp. 340–353.

[36] B. Randell, "System structure for software fault tolerance," in *Proceedings of the international conference on Reliable software.* New York, NY, USA: Association for Computing Machinery, Apr. 1975, pp. 437–449.

[37] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: A secretless framework for security through diversity." in *USENIX Security Symposium*, 2006, pp. 105–120.

[38] F. B. Cohen, "Operating system protection through program evolution." *Comput. Secur.*, vol. 12, no. 6, pp. 565–584, 1993.

[39] S. Forrest, A. Somayaji, and D. Ackley, "Building diverse computer systems," in *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No.97TB100133)*, May 1997, pp. 67–72.

[40] B. Persaud, B. Obada-Obieh, N. Mansourzadeh, A. Moni, and A. Somayaji, "Frankenssl: Recombining cryptographic libraries for software diversity," in *Proceedings of the 11th Annual Symposium On Information Assurance. NYS Cyber Security Conference*, 2016, pp. 19–25.

[41] N. Harrand, T. Durieux, D. Broman, and B. Baudry, "Automatic diversity in the software supply chain," *arXiv preprint arXiv:2111.03154*, 2021.

[42] D. Brumley and D. Boneh, "Remote timing attacks are practical," *Computer Networks*, vol. 48, no. 5, pp. 701–716, 2005.

[43] D. J. Bernstein, "Cache-timing attacks on aes," 2005.

[44] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: The Case of AES," in *Topics in Cryptology – CT-RSA 2006*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 1–20.

[45] G. Barthe, S. Blazy, R. Hutin, and D. Pichardie, "Secure Compilation of Constant-Resource Programs," in *CSF 2021 - 34th IEEE Computer Security Foundations Symposium*. IEEE, Jun. 2021, pp. 1–12.

[46] V. C. Ngo, M. Dehesa-Azuara, M. Fredrikson, and J. Hoffmann, "Verifying and Synthesizing Constant-Resource Implementations with Types," in *2017 IEEE Symposium on Security and Privacy (SP)*, May 2017, pp. 710–728, iSSN: 2375-1207.

[47] M. A. Shelton, N. Samwel, L. Batina, F. Regazzoni, M. Wagner, and Y. Yarom, "Rosita: Towards Automatic Elimination of Power-Analysis Leakage in Ciphers," *Proceedings 2021 Network and Distributed System Security Symposium*, 2021, appears in NDSS 2022.

[48] K. Ngo, E. Dubrova, and T. Johansson, "Breaking Masked and Shuffled CCA Secure Saber KEM by Power Analysis," in *Proceedings of the 5th Workshop on Attacks and Solutions in Hardware Security*, Nov. 2021, pp. 51–61.

[49] W.-J. van Hoeve and I. Katriel, "Global constraints," in *Foundations of Artificial Intelligence*. Elsevier, 2006, vol. 2, pp. 169–208.

[50] J.-C. Régin, "A filtering algorithm for constraints of difference in CSPs," in *AAAI*, vol. 94, 1994, pp. 362–367.

[51] L. Ingmar, M. Garcia de la Banda, P. J. Stuckey, and G. Tack, "Modelling diversity of solutions," in *Proceedings of the thirty-fourth AAAI conference on artificial intelligence*, 2020.

[52] E. Hebrard, B. Hnich, B. O'Sullivan, and T. Walsh, "Finding Diverse and Similar Solutions in Constraint Programming," in *National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference*, 2005, p. 6.

[53] L. Popovic, A. Côté, M. Gaha, F. Nguewouo, and Q. Cappart, "Scheduling the equipment maintenance of an electric power transmission network using constraint programming," in *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[54] M. Gendreau, J.-Y. Potvin *et al.*, *Handbook of metaheuristics*. Springer, 2010, vol. 2.

[55] P. Shaw, "Using constraint programming and local search methods to solve vehicle routing problems," in *Principles and Practice of Constraint Programming*, ser. Lecture Notes in Computer Science, vol. 1520. Springer, 1998, pp. 417–431.

[56] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*. IEEE, 2004.

[57] R. M. Stallman, *Using the GNU Compiler Collection: a GNU manual for GCC version 4.3.3*. CreateSpace, 2009.

[58] G. Hjort Blindell, *Instruction Selection*. Springer International Publishing, 2016.

[59] R. C. Lozano and C. Schulte, "Survey on Combinatorial Register Allocation and Instruction Scheduling," *ACM Computing Surveys (CSUR)*, vol. 52, no. 3, pp. 62:1–62:50, 2019.

[60] G. P. Farina, S. Chong, and M. Gaboardi, "Relational Symbolic Execution," in *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*, ser. PPDP '19. Association for Computing Machinery, Oct. 2019, pp. 1–14.

[61] H. Winderix, J. T. Mühlberg, and F. Piessens, "Compiler-Assisted Hardening of Embedded Software Against Interrupt Latency Side-Channel Attacks," in *2021 IEEE European Symposium on Security and Privacy (EuroS P)*, Sep. 2021, pp. 667–682.

[62] F. Besson, A. Dang, and T. Jensen, "Information-Flow Preservation in Compiler Optimisations," in *2019 IEEE 32nd Comput. Secur. Found. Symp. CSF*, 2019, pp. 230–23 012.

[63] T. Edgar and D. Manz, *Research methods for cyber security*. Syngress, 2017.

[64] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems," in *MICRO*. IEEE, 1997, pp. 330–335.

[65] *CPU 2006 Benchmarks*, SPEC, 2020, https://www.spec.org/cpu2006, accessed on 2020-03-20.

[66] Libsodium Community, "The sodium cryptography library (Libsodium)," 2018. [Online]. Available: https://libsodium.gitbook.io/doc

[67] J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan, "Formally Verified Cryptographic Web Applications in WebAssembly," in *2019 IEEE Symposium on Security and Privacy (SP)*, May 2019, pp. 1256–1274.

[68] T. Pornin, "Bearssl, a smaller SSL/TLS library," last accessed May 14, 2021. [Online]. Available: https://bearssl.org/

[69] H. Mantel and A. Starostin, "Transforming Out Timing Leaks, More or Less," in *Computer Security – ESORICS 2015*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 447–467.

[70] A. Lindner, R. Guanciale, and M. Dam, "Proof-producing symbolic execution for binary code verification," 2023.

[71] D. Broman, "A Brief Overview of the KTA WCET Tool," Dec. 2017, number: arXiv:1712.05264 arXiv:1712.05264 [cs].

[72] R. M. Tsoupidi, "Two-phase WCET analysis for cache-based symmetric multiprocessor systems," Master's thesis, Royal Institute of Technology KTH, 2017.

[73] S. Pastrana, J. Tapiador, G. Suarez-Tangil, and P. Peris-López, "AVRAND: A Software-Based Defense Against Code Reuse Attacks for AVR Embedded Devices," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. Lecture Notes in Computer Science, 2016, pp. 58–77.

[74] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, "C-FLAT: Control-Flow Attestation for Embedded Systems Software," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, Oct. 2016, pp. 743–754.

[75] T. Nyman, J.-E. Ekberg, L. Davi, and N. Asokan, "CFI CaRE: Hardware-Supported Call and Return Enforcement for Commercial Microcontrollers," in *Research in Attacks, Intrusions, and Defenses*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2017, pp. 259–284.

[76] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis, "Compiler-Assisted Code Randomization," in *2018 IEEE Symposium on Security and Privacy (SP)*, May 2018, pp. 461–477.

[77] J. Shi, L. Guan, W. Li, D. Zhang, P. Chen, and P. Chen, "HARM: Hardware-assisted continuous re-randomization for microcontrollers," in *2022 IEEE european symposium on security and privacy (EuroS P)*, 2022.

[78] A. Fu, W. Ding, B. Kuang, Q. Li, W. Susilo, and Y. Zhang, "FH-CFI: Fine-grained hardware-assisted control flow integrity for ARM-based IoT devices," *Computers & Security*, vol. 116, p. 102666, May 2022.

[79] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*, ser. CCS '04. Association for Computing Machinery, Oct. 2004, pp. 298–307.

[80] S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala, and D. Stefan, "FaCT: A Flexible, Constant-Time Programming Language," in *2017 IEEE Cybersecurity Dev. SecDev*, 2017, pp. 69–76.

[81] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "HACL*: A Verified Modern Cryptographic Library," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1789–1806.

[82] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P.-Y. Strub, "Jasmin: High-Assurance and High-Speed Cryptography," in *Proc. 2017 ACM SIGSAC Conf. Comput. Commun. Secur.*, ser. CCS '17. Association for Computing Machinery, 2017, pp. 1807–1823.

[83] P. Borrello, D. C. D'Elia, L. Querzoni, and C. Giuffrida, "Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization," *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pp. 715–733, Nov. 2021.

[84] J. Cabrera Arteaga, O. Floros, O. Vera Perez, B. Baudry, and M. Monperrus, "Crow: Code diversification for webassembly," in *MADWeb, NDSS 2021*, 2021.

[85] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *25th USENIX security symposium (USENIX security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 53–70.

[86] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, Jul. 2009.

[87] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson, "Vale: Verifying {High-Performance} Cryptographic Assembly Code," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 917–934.

[88] I. Bastys, M. Algehed, A. Sjösten, and A. Sabelfeld, "Secwasm: Information flow control for webassembly," in *Static Analysis: 29th International Symposium, SAS 2022, Auckland, New Zealand, December 5–7, 2022, Proceedings*. Springer, 2022, pp. 74–103.

[89] H. Eldib and C. Wang, "Synthesis of Masking Countermeasures against Side Channel Attacks," in *Comput. Aided Verification*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 114–130.

[90] P. Gao, J. Zhang, F. Song, and C. Wang, "Verifying and Quantifying Side-channel Resistance of Masked Software Implementations," *ACM Transactions on Software Engineering and Methodology*, vol. 28, no. 3, pp. 16:1–16:32, 2019.

[91] S. T. Vu, A. Cohen, K. Heydemann, A. de Grandmaison, and C. Guillon, "Secure optimization through opaque observations," *arXiv preprint arXiv:2101.06039*, 2021.

[92] F. Besson, A. Dang, and T. Jensen, "Securing Compilation Against Memory Probing," in *Proc. 13th Workshop Program. Lang. Anal. Secur.*, ser. PLAS '18. Association for Computing Machinery, 2018, pp. 29–40.

[93] L.-A. Daniel, S. Bardin, and T. Rezk, "Hunting the haunter-efficient relational symbolic execution for spectre with haunted relse," in *NDSS 2021-Network and Distributed Systems Security*, 2021.