Analysis and Enforcement of Properties in Software Systems

Meng Wu

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Patrick R. Schaumont, Co-chair
Chao Wang, Co-chair
Michael S. Hsiao
Haibo Zeng
Changhee Jung
Na Meng

April 23, 2019
Blacksburg, Virginia

Keywords: Shield Synthesis, Program Analysis, Timing Side Channel, Cache Timing Leak,
Speculative Execution

Analysis and Enforcement of Properties in Software Systems

Meng Wu

(ABSTRACT)

Due to the lack of effective techniques for detecting and mitigating property violations, existing approaches to ensure the safety and security of software systems are often labor intensive and error prone. Furthermore, they focus primarily on functional correctness of the software code while ignoring micro-architectural details of the underlying processor, such as cache and speculative execution, which may undermine their soundness guarantees.

To fill the gap, I propose a set of new methods and tools for ensuring the safety and security of software systems. Broadly speaking, these methods and tools fall into three categories. The first category is concerned with static program analysis. Specifically, I develop a novel *abstract interpretation* framework that considers both speculative execution and a cache model, and is guaranteed to be sound for estimating the execution time of a program and detecting side-channel information leaks. The second category is concerned with static program transformation. The goal is to eliminate side channels by equalizing the number of CPU cycles and the number of cache misses along all program paths for all sensitive variables. The third category is concerned with runtime safety enforcement. Given a property that may be violated by a reactive system, the goal is to synthesize an enforcer, called a *shield*, to correct the erroneous behaviors of the system instantaneously, so that the property is always satisfied by the combined system. I develop techniques to make the shield practical by handling both *burst error* and *real-valued* signals.

The proposed techniques have been implemented and evaluated on realistic applications to demonstrate their effectiveness and efficiency.

Analysis and Enforcement of Properties in Software Systems

Meng Wu

(GENERAL AUDIENCE ABSTRACT)

It is important for everything around us to follow some rules to work correctly. That is the same for our software systems to follow the security and safety properties. Especially, softwares may leak information via unexpected ways, e.g. the program timing, which makes it more difficult to be detected or mitigated. For instance, if the execution time of a program is related to the sensitive value, the attacker may obtain information about the sensitive value. On the other side, due to the complexity of softwares, it is nearly impossible to fully test nor verify them. However, the correctness of software systems at runtime is crutial for critical applications.

While exsiting approaches to find or resolve properties violation problem are often labor intensive and error prone, in this dissertation, I first propose an automated tool for detecting and mitigating the security vulnerability through program timing. Programs processed by the tool are guaranteed to be time constant under any sensitive values. I have also taken the influence of speculative execution, which is the cause behind recent Spectre and Meltdown attack, into consideration for the first time. To enforce the correctness of programs at runtime, I introduce an extra component that can be attached to the original system to correct any violaton if it happens, thus the entire system will still be correct.

All proposed methods have been evaluated on a a variety of real world applications. The results show that these methods are effective and efficient in practice.

Acknowledgments

It would have been impossible for me to complete the research and write this dissertation without the help of many people.

First and foremost, I want to thank my dissertation advisor, Dr. Chao Wang. He is an amazing scholar and his pursuit of excellence in both research and teaching has made a significant impact on how I approach work and life.

I want to thank my co-advisor, Dr. Patrick Schaumont, for his help and guidance on research. Conversations with him have always been pleasant and inspiring.

It is a great honor and privilege to have Dr. Michael Hsiao, Dr. Haibo Zeng, Dr. Changhee Jung, and Dr. Na Meng on my dissertation committee. I have benefited a lot from their teaching and research advice.

Looking back, my journey in the past five years has been filled with joy, but sometimes it can feel like grinding. I want to thank my wife for her endless love and support, without which I would not have been able to finish.

Contents

List of Figures xi

List of Tables xiv

1 Introduction 1

1.1 Overview 5

1.2 Contribution 6

1.3 Outline 8

2 Detecting Side Channel Leaks 10

2.1 Timing Side Channels 10

2.1.1 Conditional Jumps Affected by Secret Data 10

2.1.2 Table Lookups Affected by Secret Data 12

2.1.3 Idiosyncratic Code Affected by Secret Data 15

2.2 Threat Model 16

2.3 Detecting Timing Leaks 17

2.3.1 Static Sensitivity Analysis 17
2.3.2 Leaky Conditional Statements 19
2.3.3 Leaky Lookup-table Accesses 19

3 Mitigating Side Channel Leaks **20**

3.1 Mitigating Conditional Statements 20
3.1.1 Standardizing Conditional Statements 21
3.1.2 Replacing Conditional Statements 23
3.1.3 Optimizations 24
3.2 Mitigating Lookup Table Accesses 25
3.2.1 Mitigation Granularity and Overhead 26
3.2.2 Static Cache Analysis 27
3.2.3 Static Cache Analysis-based Reduction 32
3.3 Related Work 33

4 Speculative Cache Analysis **36**

4.1 Introduction to Speculative Execution 37
4.2 Cache Analysis under Speculative Execution 38
4.2.1 Execution Time Estimation 39
4.2.2 Side Channel Detection 40
4.3 Technical Challenges 41

4.4 Preliminaries . . . 42
4.4.1 Abstract Interpretation . . . 42
4.4.2 Cache and Speculative Execution . . . 43
4.5 Modeling the Speculative Execution . . . 44
4.5.1 Augmented CFG with Virtual Control Flow . . . 45
4.5.2 Merging the Speculative Flows . . . 47
4.5.3 Just-in-Time Merging: An Example . . . 48
4.6 Generalization and Optimization . . . 49
4.6.1 The Running Example . . . 50
4.6.2 Dynamically Bounding Speculation Depth . . . 54
4.6.3 Handling the Merges and Loops . . . 55
4.6.4 Handling Multiple Speculative Executions . . . 57
4.7 Related Work . . . 58

5 Runtime Enforcement under Burst Error 60

5.1 Introduction to the Shield . . . 60
5.1.1 The Reactive System . . . 60
5.1.2 The Safety Shield . . . 62
5.1.3 Example of Shield Handling Burst Error . . . 63
5.2 Synthesize Shield under Burst Error . . . 65

5.2.1 The Overall Flow . . . 66
5.2.2 Constructing the Safety Game . . . 67
5.3 Solving the Safety Game . . . 73
5.3.1 Fix-point Computation . . . 74
5.3.2 Optimization . . . 74
5.4 Related Work . . . 75

6 Runtime Enforcement for Real-Valued Signals 77

6.1 Technical Challenges . . . 79
6.1.1 Realizability of the Boolean Predicates . . . 79
6.1.2 Quality of the Real-valued Output . . . 80
6.2 Synthesizing the Boolean Shield . . . 81
6.2.1 Computing the Predicates . . . 83
6.2.2 Computing the Boolean Abstractions . . . 84
6.2.3 Computing the Relaxation Automaton . . . 84
6.2.4 Computing the Feasibility Automaton . . . 86
6.2.5 Solving the New Safety Game . . . 86
6.3 Generating the Real-valued Signals . . . 87
6.3.1 Robustness Optimization . . . 88
6.3.2 Value Prediction and Validation . . . 89

6.4 Related Work 90

7 Evaluation **92**

7.1 Timing Side Channel Elimination 92

7.1.1 Benchmarks 93

7.1.2 Experimental Results: Leak Detection 93

7.1.3 Experimental Results: Leak Mitigation 96

7.1.4 Experimental Results: Simulation 97

7.1.5 Threats to Validity 99

7.2 Cache Analysis under Speculative Execution 100

7.2.1 Benchmarks 100

7.2.2 Effectiveness: Execution Time Estimation 101

7.2.3 Effectiveness: Side Channel Detection 103

7.3 Boolean Shield under Burst Error 104

7.3.1 Benchmarks 105

7.3.2 Experimental Results 105

7.4 Real-Valued Shield Synthesis 108

7.4.1 Benchmarks 108

7.4.2 Experimental Results 109

7.4.3 Case Studies 111

8 Conclusions **114**

Bibliography **116**

List of Figures

1.1 Detecting and mitigating both *instruction*- and *cache*-timing side channels. 6

1.2 Overview of the safety shield for real. 7

2.1 A cipher with timing leaks and two different mitigation approaches. 11

2.2 Example for accessing the lookup table. 13

2.3 Countermeasure: reading all the elements. 13

2.4 Countermeasure: reading all cache lines. 14

2.5 Countermeasure: preloading all cache lines. 14

2.6 RC5.c 15

2.7 Example of Field Sensitive Pointer Analysis 18

3.1 A not-yet-standardized conditional statement. 22

3.2 Standardized conditional statements (Fig. 3.1). 22

3.3 Removing the conditional jumps. 25

3.4 Reduction: preloading only in the first iteration. 28

3.5 Transfer of the cache state under the LRU policy. 30

3.6 Update of the abstract cache state: (1) on the left-hand side, join at the merge point of two paths; and (2) on the right-hand side, a non-deterministic key for memory access. 30

4.1 Example program for timing side channel. 39

4.2 Pipelined execution trace for program in Figure 4.1 40

4.3 Strategies for merging speculative control flows. 46

4.4 Cache state with different merge points. 48

4.5 Code snippet from a real-time DSP program [76]. 51

4.6 Augmented CFG with virtual control flows. 52

4.7 The client code that leads to side-channel leaks. 53

4.8 Example program for the widening operator. 56

4.9 Transfer function with shadow variables. 57

4.10 The refined join using shadow variables. 57

5.1 Example safety specification φ^s. 64

5.2 The 2-stabilizing shield [31]. 64

5.3 Our new shield for burst error. 64

5.4 Simulation trace of 2-stabilizing shield. 65

5.5 Simulation trace of our new shield. 65

5.6 Example: (a) safety specification $\varphi^s(R, S)$ and (b) correctness monitor $\mathcal{Q}(R, S')$. . 68

5.7 Constructing the violation monitor $\mathcal{U}(R, S)$: Replacing edge $1 \rightarrow 2$ with $1 \rightarrow \{0, 1\}$. 71

5.8 Violation monitor $\mathcal{U}(R, S)$. 72

5.9 Deviation monitor $\mathcal{T}(S, S')$. 72

5.10 Error-avoiding monitor $\mathcal{E}(R, S, S')$. 72

5.11 Game graph $\mathcal{G}(R, S, S')$, which is the composition of $\mathcal{Q}(R, S')$ and $\mathcal{E}(R, S, S')$. . . 73

6.1 Importance of the smoothness in real-valued correction signals. 81

6.2 Relaxation automaton $\mathcal{R}(I, O)$: *impossible* means the system $\mathcal{D}$ will not allow the state to be reached, and the shield $\mathcal{S}$ can treat it as *don't care*. 85

6.3 Feasibility automaton $\mathcal{F}(O')$: *infeasible* means the state is unrealizable, and the shield $\mathcal{S}$ must avoid the related edges while generating solutions. 85

7.1 Reduction: preloading only in the first iteration. 96

7.2 *CTSEL* implementations. 98

7.3 Automotive powertrain system simulation results (with and without the shield). . . 112

7.4 Position, velocity and acceleration in autonomous driving simulation. 113

List of Tables

3.1 Overhead comparison: N is the table size; $M = \lceil N/CLS \rceil$ is the number of cache lines to store the table; K is the number of times table elements are accessed. . . . 27

4.1 Cache states during the fixed-point computation. 52

4.2 Cache states during speculative execution. 53

7.1 Benchmark statistics. 94

7.2 Results of conducting static leakage detection. 95

7.3 Results of leakage mitigation. Runtime overhead is based on average of 1000 simulations with random keys. 97

7.4 Results of GEM5 simulation with 2 random inputs. 99

7.5 Execution time estimation: benchmark statistics. 101

7.6 Side channel detection: benchmark statistics. 101

7.7 Execution time estimation: comparisons in terms of the analysis time and the number of cache misses. 102

7.8 Execution time estimation: comparisons of two strategies for merging speculative executions. 103

7.9 Side channel detection: comparisons in terms of the analysis time and whether leaks are detected. 104

7.10 Experimental results for comparing the two shield synthesis algorithms. 106

7.11 Experimental results for synthesizing the shield with and without optimization. . . 107

7.12 Statistics of the benchmark applications. 109

7.13 Results of real shield synthesis procedure. 110

7.14 Results of evaluating runtime performance of the shield. 111

Chapter 1

Introduction

Ensuring the safety and security of software systems is a grand challenge of our time. However, it becomes more and more difficult to guarantee functional correctness as the complexity of software keeps increasing. Furthermore, non-functional properties of the hardware, such as the power consumption of the computing device and execution time of the program, start to have a significant impact on the security of the software. For example, while timing side channels have long been exploited by attackers to deduce secret information of the system, such as cryptographic keys, passwords, and security tokens, recent attacks such as Meltdown, Spectre, and ForeShadow [85, 98, 156] showed more severe threats to a much broader class of systems and applications.

While conventional techniques for software testing and verification are invaluable, they are known to have either coverage or scalability problems, and therefore cannot be guaranteed to eliminate all safety violations and security vulnerabilities in practice. In particular, they are not effective in detecting or mitigating side-channel leaks, which can be exploited by an adversary easily. For example, many modern processors have performance-related instructions for measuring the CPU usage, which allow attackers to easily monitor the local processes [100, 116, 157] or measure the response time of remote servers. At the same time, mitigating the timing side channel is difficult

since it depends on low-level compiler optimizations and even hardware components inside the CPU, such as instruction and data caches [11], pipelines, and the branch predictors [12].

Since it is not always possible to eliminate safety and security violations at design time, there is also a need to enforce critical properties at run time. To avoid the same scalability problems encountered by conventional testing and verification techniques, construction and execution of these runtime enforcers should not depend on the source code and other implementation details of the potentially complex system. Ideally, they should depend only on the (hopefully small set of) critical properties to be enforced. Unfortunately, such runtime enforcement approaches are still severely lacking.

To fill the gap, I develop a set of new methods and tools for ensuring the safety and security of critical software systems. In terms of security, we address the problem of detecting and mitigating side-channel information leaks by developing a novel abstract interpretation based static analysis framework, followed by compiler-based program transformations. In terms of safety, we address the problem by automatically synthesizing a runtime enforcer, called a shield, to enforce a set of safety-critical properties of a reactive system.

In the first line of work, we are concerned with static program analysis and transformation techniques, and the goal is to mitigate two types of timing side-channel leaks: instruction-related and cache-related. By *instruction*-related, we mean the number or type of instructions executed along an execution trace may differ depending on the values of secret variables, leading to differences in the number of CPU cycles; By *cache*-related, we mean the memory subsystem may behave differently depending on the values of secret variables, e.g., a cache hit takes few CPU cycles but a miss takes hundreds of cycles.

Manually analyzing the timing characteristics of software code is difficult because it requires deep knowledge of not only the application itself but also the micro-architecture of the computer, including the cache configuration and how software code is compiled to machine code. Even if our

heroic programmer is able to conduct the aforementioned timing analysis manually, it would be too labor-intensive and error-prone to be economical in practice: with every code change, the software has to be re-analyzed and countermeasures have to be re-applied to ensure uniform execution time for all possible values of the secret variables.

It is worth noting that straightforward countermeasures such as noise injection (i.e., adding random delay to the execution) do not work well in practice, because noise can be removed using well-established statistical analysis techniques [86, 87].

Thus, we propose a fully automated method for mitigating the timing side channels. Our method relies on static analysis to identify, for a program and a list of *secret* inputs, the set of variables whose values depend on the secret data. To decide if these *sensitive* variables lead to timing leaks, we check if they affect unbalanced conditional jumps (instruction-timing leaks) or accesses of memory blocks spanning across multiple cache lines (cache-timing leaks).

Based on results of this analysis, we perform code transformations to mitigate the leaks, by equalizing the execution time. Conceptually, these transformations are straightforward: if we equalize the execution time of both sensitive conditional statements and sensitive memory accesses, there will be no instruction- or cache-timing leaks.

However, since both transformations adversely affect the runtime performance, they must be applied judiciously to remain practical. Thus, a main technical challenge is to develop analysis techniques to decide *when* these countermeasures are *not needed* and thus can be skipped safely.

A *static cache analysis* is used to identify the set of locations where memory accesses always lead to cache hits. This *must-hit* analysis, following the unified framework of abstract interpretation [44], is designed to be conservative in that a reported must-hit is guaranteed to be a hit along all paths. Thus, it can be used by our tool to skip redundant mitigations. Unfortunately, existing abstract interpretation techniques [51, 65, 67, 143] are unsound under speculative execution. Instead, these

prior works on abstract interpretation focus more on modeling *non-speculative* executions, for which numerous techniques have been developed, including widening/narrowing, chaotic iteration, and efficient implementations of abstract domains.

Under *speculative* execution, however, none of these techniques are relevant because the problem is no longer about removing infeasible paths from the over-approximated analysis, but about preventing real behaviors from being excluded. This requires a different set of ideas from what already exist in the literature. The need for a sound cache analysis was highlighted by attacks such as Spectre [85], Meltdown [98] and Foreshadow [156]. Thus, I propose a method for lifting the abstract interpretation framework so that it becomes sound again under speculative execution.

The aforementioned cache timing side channel detection and mitigation techniques have been implemented in software tools and evaluated on various cryptographic libraries and Linux kernel modules. The results of our evaluation show only moderate increases in the program code size and the runtime overhead. At the same time, the mitigated software programs are guaranteed to be side-channel leak free. We also confirm, using a micro-architectural simulator named GEM5, that the mitigated programs generated by our tools are indeed leakage free.

In the second line of work, I extend the original shield synthesis algorithm of Bloem et al. [31], to handle burst error and real-valued signals. At a high level, a shield $\mathcal{S}$ is a runtime enforcer of a safety property φ of a reactive system $\mathcal{D}$. That is, regardless of the runtime behavior of $\mathcal{D}$, it ensures that the combined $\mathcal{D} \circ \mathcal{S}$ never violates φ. If, for example, $\mathcal{D}(I, O)$ malfunctions and produces some erroneous output O for input I, the shield will correct O into O' instantaneously to ensure $\varphi(I, O')$ holds even when $\varphi(I, O)$ fails. An important feature of the shield is that $\mathcal{S}$ is synthesized solely from φ, regardless of the internals of $\mathcal{D}$, which makes it well-suited for systems with arbitrarily complex $\mathcal{D}$ but small φ, e.g., learning-enabled systems [17, 155, 164].

An important feature of the shield is that it tries to minimize the deviation between the original and

modified system output. However, the original shield synthesis algorithm of Bloem et al. does not robustly handle burst error; instead, it allows only one error to occur within a window of k time steps. If there are more than one errors within k time steps, the shield would enter a fail-safe state, from which it stops minimizing the deviation. We extend the synthesis algorithm to make sure that the shield can handle arbitrary errors without ever going to the fail-safe state.

Another problem of the original shield synthesis algorithm of Bloem et al. is that the shield can only handle Boolean signals. However, in many real world systems, the input and output have real-valued signals. Directly applying the Boolean shield to real-valued signals may run into both realizability and scalability problems. Thus, I develop a new shield synthesis algorithm, to extend the shield from the Boolean domain to the real-valued domain, where the system's input and output signals can have real values.

The proposed shield synthesis algorithms have been implemented in software tools, and the resulting shields have been evaluated on a number of embedded control applications. Our experiments show that, in all cases, the shields synthesized by our new methods are significantly more effective in handling burst error as well as producing real-valued correction signals at run time.

1.1 Overview

Figure 1.1 shows the overall flow of the software tool that we implement for detecting and mitigating side-channel information leaks. Given the original program written in the C language, first, our tool parses the program to construct its intermediate representation in LLVM. Then, it conducts a series of static analyses to identify the *sensitive* variables and timing leaks associated with these variables. Next, it performs two types of code transformations to remove the leaks. One transformation aims to eliminate the differences in the execution time caused by unbalanced conditional jumps, while the other transformation aims to eliminate the differences in the number of cache hits/misses during

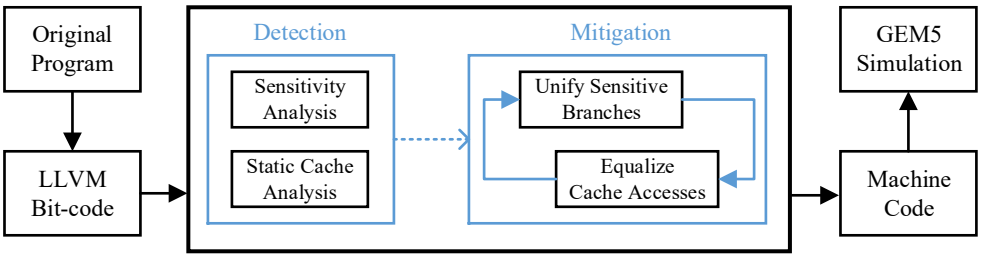


Figure 1.1: Detecting and mitigating both *instruction*- and *cache*-timing side channels.

the accesses of look-up tables such as S-Boxes. Finally, the GEM5 simulator is used to confirm that the mitigated program code is indeed leakage free.

Fig. 1.2 represents the overall flow of our synthesized shield, where the input consists of real-valued I_r and O_r signals and a safety property φ_r defined over these signals. Internally, the shield $\mathcal{S}$ has three subcomponents: a converter from real-valued I_r/O_r signals to Boolean I/O signals, a converter from Boolean O' signals to real-valued O'_r signals, and a Boolean shield $\mathcal{S}(I, O, O')$. The synthesized shield is meant to be attached to the original system, to monitor its input and instantaneously correct the erroneous output. That is, if the system satisfies the safety property, the shield's output will remain the same as the system's output; but if the system violates the safety property, the shield will take action immediately by generating some new output, to satisfy the safety property.

1.2 Contribution

In terms of practical contributions, the methods and tools developed in this dissertation are fully automated and can be used to significantly improve the safety and security of software systems. Specifically, they can detect timing side-channel leaks using static program analysis, mitigating these leaks using program transformations, and finally enforce safety properties of systems at run

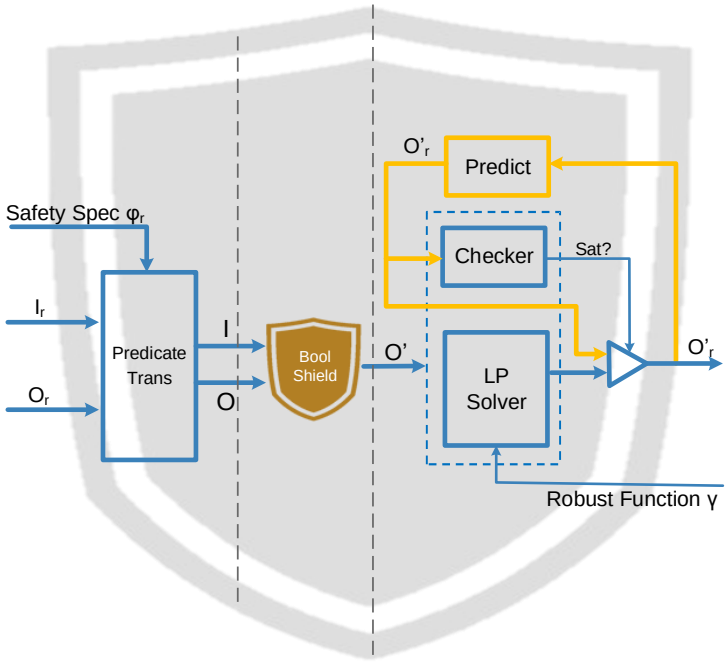


Figure 1.2: Overview of the safety shield for real.

time.

In terms of scientific contributions, the new abstract interpretation based static program analysis framework developed in this dissertation is the first such technique for soundly handling speculative execution. The new algorithms for synthesizing the shields are also the first for robustly handling burst error and producing real-valued correction signals.

To summarize, this dissertation makes the following contributions:

- I propose a static analysis and transformation based method for eliminating instruction- and cache-timing side channels.

- I propose a generally applicable abstract interpretation framework to make the analysis sound under speculative execution. The speculative cache analysis can help improve side channel mitigation safely.

- I propose a method for synthesizing the shield while minimizing the deviation under burst error.

- I propose a method for synthesizing the shield to ensure both the realizability and the efficiency of the shield in generating real-valued correction signals.

- I demonstrate, using realistic benchmark applications, the effectiveness and efficiency of the proposed techniques.

1.3 Outline

The remainder of this dissertation is organized as follows:

Chapter 2 presents a number of static program analysis techniques for detecting side-channel leaks, including a sensitivity analysis, which propagates a set of user-provided annotations from inputs to other program variables.

Chapter 3 presents a number of program transformation techniques for mitigating side-channel leaks. They are provably secure in eliminating leaks associated with both conditional jumps and lookup table accesses. These mitigation techniques are further optimized using a static cache analysis, which reduces the unnecessary mitigation points where instructions cannot actually cause leaks.

Chapter 4 presents a new method for making the cache analysis presented in Chapter 3 sound under speculative execution. That is, it is guaranteed to capture the cache behaviors along all program paths, for all program inputs, and under all speculative executions.

Chapter 5 presents a new method for synthesizing the shield to handle burst error. Toward this end, we first review the basics of the classic algorithm for shield synthesis, and then present our extension to handle burst error.

Chapter 6 presents a further extension of the synthesis algorithm presented in Chapter 5, to handle real-valued input and output signals of the system. We also illustrate the effectiveness of the proposed technique using two case studies in the context of realistic control systems.

Chapter 7 presents the evaluation results, which demonstrate the effectiveness of all techniques presented in this dissertation.

Finally, Chapter 8 presents the conclusions.

Chapter 2

Detecting Side Channel Leaks

2.1 Timing Side Channels

The execution time of a program has been expoited attackers to deduce sensitive information such as cryptographic keys and passwords. In this section, we use examples to illustrate various types of timing leaks and then present our static analysis based methods for detecting them.

2.1.1 Conditional Jumps Affected by Secret Data

An unbalanced if-else statement whose condition is affected by secret data may have side-channel leakage, because the *then*- and *else*-branches will have different execution time. Figure 2.1 shows the C code of the textbook implementation of a `3-way` cipher [142], where the variable a is marked as secret and it affects the execution time of the if-statements. By observing the timing variation, an adversary may be able to gain information about the bits of a.

To remove the dependencies between execution time and secret data, one widely-used approach

```
1  void mu(int32_t *a) {              // original version
2    int i;
3    int32_t b[3];
4    b[0] = b[1] = b[2] = 0;
5    for (i=0; i<32; i++) {
6      b[0] <<= 1; b[1] <<= 1; b[2] <<= 1;
7      if(a[0]&1) b[2] |= 1;       // leak
8      if(a[1]&1) b[1] |= 1;       // leak
9      if(a[2]&1) b[0] |= 1;       // leak
10     a[0] >>= 1; a[1] >>= 1; a[2] >>= 1;
11   }
12   a[0] = b[0]; a[1] = b[1]; a[2] = b[2];
13 }
```

```
1 // mitigation #1: equalizing the branches
2 int32_t dummy_b[3];
3 dummy_b[0] = dummy_b[1] = dummy_b[2] = 0;
4 ...
5   dummy_b[0] <<= 1; dummy_b[1] <<= 1; dummy_b[2] <<= 1;
6   ...
7   if(a[0]&1) b[2]|=1; else dummy_b[2]|=1;
8   if(a[1]&1) b[1]|=1; else dummy_b[1]|=1;
9   if(a[2]&1) b[0]|=1; else dummy_b[0]|=1;
```

```
1 // mitigation #2: removing the branches
2 b[2] = CTSEL(a[0]&1, b[2]|1, b[2]);
3 b[1] = CTSEL(a[1]&1, b[1]|1, b[1]);
4 b[0] = CTSEL(a[2]&1, b[0]|1, b[0]);
```

Figure 2.1: A cipher with timing leaks and two different mitigation approaches.

is equalizing the branches by cross-copying as illustrated by the code snippet in the middle of Figure 2.1: the auxiliary variable `dummy_b[3]` and some assignments are added to make both branches have the same number and type of instructions. Unfortunately, this approach does not always work in practice, due to the presence of hidden states at the micro-architectural level and related performance optimizations inside modern CPUs (e.g., instruction caching, branch prediction and speculative execution) – we have confirmed this limitation by analyzing the mitigated code using GEM5, the details of which are described as follows.

We compiled the mitigated program shown in the middle of Figure 2.1 and, by carefully inspecting

the machine code, made sure that all conditional branches indeed had the same number (and type) of instructions. Then, we ran the top-level program on GEM5 with two different cryptographic keys: k_1 has 1's in all 96 bits whereas k_2 has 0's in all 96 bits. Our GEM5 simulation results showed significant timing differences: 88,014 CPU cycles for k_1 versus 87,624 CPU cycles for k_2. Such timing variations would allow attackers to gain information about the secret key.

Therefore, in the remainder of this paper, we avoid the aforementioned approach while focusing on an alternative: *replacing sensitive branches* with functionally-equivalent, constant-time, branch-less assignments, as shown in the code snippet at the bottom of Figure 2.1. Specifically, *CTSEL(c,t,e)* is an LLVM intrinsic we added to ensure the selection of either t or e, depending on the predicate c, is done in constant time. For different CPU architectures, this intrinsic function will be compiled to different machine codes to obtain the best performance possible (see Section 3.1 for details). Because of this, our mitigation adds little runtime overhead: the mitigated program requires only 90,844 CPU cycles for both k_1 and k_2.

Note that we cannot simply rely on C-style conditional assignment `r=(c?t:e)` or the LLVM *select* instruction because neither guarantees constant-time execution. Indeed, LLVM may transform both to conditional jumps, e.g., when r is of `char` type, which may have the same residual timing leaks as before. In contrast, our use of the new *CTSEL* intrinsic avoids the problem.

2.1.2 Table Lookups Affected by Secret Data

When an index used to access a lookup table (LUT) depends on the secret data, the access time may vary due to the behavior of the cache associated with the memory block. Such cache-timing leaks have been exploited, e.g., in block ciphers [74, 116, 148] that for efficiency reasons implement `S-Boxes` using lookup tables. Figure 2.2 shows the `subBytes` function of the AES cipher in FELICS [49], which substitutes each byte of the input array (`block`) with the precomputed

```
const uint8_t sbox[256] = {0x63,0x7c,0x77,0x7b,0xf2,0x6b,0x6f,0xc5,
  0x30,0x01,0x67,0x2b,0xfe,0xd7,0xab,0x76, ...};
void subBytes(uint8_t *block) {
  uint8_t i;
  for (i = 0; i < 16; ++i) {
    block[i] = sbox[block[i]];
  }
}
```

Figure 2.2: Example for accessing the lookup table.

```
//mitigation #3: replacing block[i] = sbox[block[i]];
block_i = block[i];
for (j=0; j < 256; j++) {
  sbox_j = sbox[j];
  val = (block_i == j)? sbox_j : block_i;
}
block[i] = val;
```

Figure 2.3: Countermeasure: reading all the elements.

byte stored in `sbox`. Thus, the content of `block`, which depends on secret data, may affect the execution time. For example, when all sixteen bytes of `block` are `0x0`, meaning `sbox[0]` is always accessed, there will be one cache miss followed by fifteen hits; but when all sixteen bytes of `block` differ from each other, there may be 256/64 = 4 cache misses (if we assume 64 bytes per cache line).

Mitigating cache-timing leaks is different from mitigating instruction-timing leaks. Generally speaking, the level of granularity depends on the threat model (i.e., what the attackers can and cannot do). For example, if we add the accesses of all elements of `sbox[256]` to each original read of `sbox[]`, as shown in Figure 2.2, it would be impossible for attackers to guess which is the desired element. Since each original loop iteration now triggers the same set of LUT accesses, there is no longer timing variation.

However, the high runtime overhead may be unnecessary, e.g., when attackers cannot observe the timing variation of each loop iteration. If, instead, the attackers can only observe differences in the cache line associated with each write to `block[i]`, it suffices to use the approach in

```
1 //mitigation #4: replacing block[i] = sbox[block[i]];
2 block_i = block[i];
3 for (j=block_i % CLS; j < 256; j+=CLS) {
4   sbox_j = sbox[j];
5   val = (block_i == j)? sbox_j : block_i;
6 }
7 block[i] = val;
```

Figure 2.4: Countermeasure: reading all cache lines.

```
1 //mitigation #5: preloading sbox[256]
2 for (j =0; j < 256; j+=CLS)
3   temp = sbox[j];
4 //access to sbox[...] is always a hit
5 for (i = 0; i < 16; ++i) {
6   block[i] = sbox[block[i]];
7 }
```

Figure 2.5: Countermeasure: preloading all cache lines.

Figure 2.4. Here, `CLS` denotes the cache line size (64 bytes in most modern CPUs). Note there is a subtle difference between this approach and the naive preloading (Figure 2.5): the latter would be vulnerable to `Flush+Reload` attacks. For example, the attackers can carefully arrange the `Flush` after `Preload` is done, and then perform `Reload` at the end of the victim's computation; this is possible because `Preload` triggers frequent memory accesses that are easily identifiable by the attacker. In contrast, the approach illustrated in Figure 2.4 can avoid such attacks.

If the attackers can only measure the total execution time of a program, our mitigation can be more efficient than Figures 2.5 and 2.4: For example, if the cache is large enough to hold all elements, preloading would incur `256/CLS`=4 cache misses, but all subsequent accesses would be hits. This approach will be illustrated in Figure 3.4. However, to safely apply such optimizations, we need to make sure the table elements never get evicted from the cache. For simple loops, this would be easy. But in real applications, loops may be complex, e.g., containing branches, other loops, and function calls, which means in general, a sound static program analysis procedure is needed to determine whether an lookup table access is a MUST-HIT.

```
typedef struct {
  uint32_t *xk; // the round keys
  int nr;       // the number of rounds
} rc5_ctx;
#define ROTL32(X,C) (((X)<<(C))|((X)>>(32-(C))))
void rc5_encrypt(rc5_ctx *c, uint32_t *data, int blocks) {
  uint32_t *d,*sk;
  int h,i,rc;
  d = data;
  sk = (c->xk)+2;
  for (h=0; h<blocks; h++) {
    d[0] += c->xk[0];
    d[1] += c->xk[1];
    for (i=0; i<c->nr*2; i+=2) {
      d[0] ^= d[1];
      rc = d[1] & 31;
      d[0] = ROTL32(d[0],rc);
      d[0] += sk[i];
      d[1] ^= d[0];
      rc = d[0] & 31;
      d[1] = ROTL32(d[1],rc);
      d[1] += sk[i+1];
    }
    d+=2;
    }
}
```

Figure 2.6: RC5.c

2.1.3 Idiosyncratic Code Affected by Secret Data

For various reasons, complex operations in cryptographic software are often implemented using a series of simpler but functionally-equivalent operations. For example, the shift operation (X<<C) may be implemented using a sensitive data-dependent loop with additions: for(i=0;i<C;i++) {X += X;} because some targets (e.g. MSP430) do not support multi-bit shifts.

One real example of such idiosyncratic code is the implementation of `rc5_encrypt` [142] shown in Figure 2.6. Here, the second parameter of `ROTL32()` is aliased to the sensitive variable `c->xk`. To eliminate the timing leaks caused by an idiosyncratic implementation of (X<<C), we must conservatively estimate the loop bound. If we know, for example, the maximum va-

lue of C is MAX_C, the data-dependent loop may be rewritten to one with a fixed loop bound. for(i=0;i<MAX_C;++i) {if(i<C) X += X;} After this transformation, we can leverage the aforementioned mitigation techniques to eliminate leaks associated with the if(i<C) statement.

2.2 Threat Model

We now define the threat model, as well as timing side-channel leaks under our threat model.

We assume a *less-capable* attacker who can only observe variations of the total execution time of the victim's program with respect to the secret data. Since the capability is easier to obtain than that of a more-capable attacker, it will be more widely applicable. A classic example, for instance, is when the victim's program runs on a server that can be probed and timed remotely by the attacker using a malicious client.

We do not consider the *more-capable* attacker who can directly access the victim's computer to observe hidden states of the CPU at the micro-architectural level, e.g., by running malicious code to perform Meltdown/Spectre [85, 98] or similar cache attacks [121, 166] (Evict+Time, Prime+Probe, and Flush+Reload). Mitigating such attacks at the software level only will likely be significantly more expensive — we leave it for future work.

Let P be a program and $in = \{X, K\}$ be the input, where X is *public* and K is *secret*. Let x and k be concrete values of X and K, respectively, and $\tau(P, x, k)$ be the time taken to execute P under x and k. We say P is free of timing side-channel leaks if

$$\forall x, k_1, k_2 : \tau(P, x, k_1) = \tau(P, x, k_2) \ .$$

That is, the execution time of P is independent of the secret input K. When P has timing leaks, on the other hand, there must exist some x, k_1 and k_2 such that $\tau(P, x, k_1) \neq \tau(P, x, k_2)$.

We assume P is a deterministic program whose execution is determined completely by the input. Let $\pi = inst_1, \ldots, inst_n$ be an execution path, and $\tau(inst_i)$ be the time taken to execute each instruction $inst_i$, where $1 \leq i \leq n$, we have $\tau(\pi) = \Sigma_{i=1}^{n} \tau(inst_i)$.

Furthermore, $\tau(inst_i)$ consists of two components: $\tau_{cpu}(inst_i)$ and $\tau_{mem}(inst_i)$, where τ_{cpu} denotes the time taken to execute the instruction itself and $\tau_{mem}(inst_i)$ denotes the time taken to access the memory. For *Load* and *Store*, in particular, $\tau_{mem}(inst_i)$ is determined by if the access leads to a cache hit or miss. For the other instructions, $\tau_{mem}(inst_i) = 0$. We want to equalize both components along all program paths – this will be the foundation of our leak mitigation technique.

2.3 Detecting Timing Leaks

Now, we present our method for detecting timing leaks, which is implemented as a sequence of LLVM passes at the IR level. It takes a set of inputs marked as *secret* and returns a set of instructions whose execution time may depend on these secret inputs.

2.3.1 Static Sensitivity Analysis

To identify the leaks, we need to know which program variables are dependent of the *secret* — they are called the *sensitive* variables. Since manual annotation is tedious and error prone, we develop a procedure to perform such annotation automatically.

Secret Source: The initial set of *sensitive* variables consists of the secret inputs marked by the user. For example, in block ciphers, the secret input would be the cryptographic key while plaintext would be considered as public.

Tag Propagation: The *sensitivity* tag is an attribute to be propagated from the secret source to other

```
1  struct aes_ctx {
2    uint32_t key_enc[60];
3    uint32_t key_length;
4  };
5  int expand_key(const uint8_t *in_key, struct aes_ctx *ctx, unsigned
       int key_len)
6  {
7    uint32_t *key =ctx->key_enc;
8    key[0] = *((uint32_t*)in_key);
9    ...
10   ctx->key_length = key_len;
11   ...
12   if (ctx->key_length)
13     ...
14 }
```

Figure 2.7: Example of Field Sensitive Pointer Analysis

program variables following either data or control dependency. An example of data dependency is the *def-use* relation in {b = a & 0x80;} where b is marked as sensitive because it depends on the most significant bit of a, the sensitive variable. An example of control dependency is in if (a==0x10) {b=1;} else {b=0;} where b is marked as sensitive because it depends on whether a is `0x10`.

Field-sensitive Analysis: To perform the sensitivity analysis defined above, we need to identify aliased expressions, e.g., syntactically-different variables or fields of structures that point to the same memory location. Cryptographic software often have this type of pointers and structures. For example, the ASE implementation of Chronos [48] shown in Figure 2.7 demonstrates the need for field-sensitivity during static analysis. Here, local pointer `key` becomes sensitive when `key[0]` is assigned the value of another sensitive variable `in_key`. Without field sensitivity, one would have to mark the entire structure as sensitive (to avoid missing potential leaks). In contrast, our method performs a field-sensitive pointer analysis [22, 125] to propagate the sensitivity tag only to truly relevant fields such as `key_enc` inside `ctx`, while avoiding fields such as `key_length`. This means we can avoid marking (falsely) the unbalanced if (ctx−>key_length) statement as leaky.

2.3.2 Leaky Conditional Statements

There are two requirements for a branch statement to have potential timing leaks. First, the condition depends on secret data. Second, the branches are unbalanced. Figure 2.1 shows an example, where the conditions depend on the secret input `a` and the branches obviously are unbalanced. Sometimes, however, even if two conditional branches have the same number and type of instructions, they still result in different execution time due to hidden micro-architectural states, as we have explained in Section 2.1 and confirmed using GEM5 simulation. Thus, to be conservative, we consider *all* sensitive conditional statements as potential leaks (regardless of whether they have balanced) and apply our *CTSEL* based mitigation.

2.3.3 Leaky Lookup-table Accesses

The condition for a lookup-table (LUT) access to leak timing information is that the index used in the access is sensitive. In practice, the index affected by secret data may cause memory accesses to be mapped to different cache lines, some of which may have been loaded and thus result in hits while others result in misses. Therefore, we consider LUT accesses indexed by sensitive variables as potential leaks, e.g., the load from `sbox` in Figure 2.2, which is indexed by a sensitive element of *block*.

However, not all LUT accesses are leaks. For example, if the table has already been loaded, the (sensitive) index would no longer cause differences in the cache. This is an important optimization we perform during leak mitigation — the analysis required for deciding *if an LUT access results in a must-hit* will be presented in Section 5.3.2.

Chapter 3

Mitigating Side Channel Leaks

In Chapter 2, we have presented our static analysis techniques for detecting side-channel leaks. In this chapter, we present our program transformations for eliminating both instruction-related and cache-related timing leaks. To reduce the mitigation overhead, we leverage a static cache analysis to reduce the number of mitigation points.

3.1 Mitigating Conditional Statements

We present our method for mitigating leaks associated with conditional jumps. In contrast to existing approaches that only attempt to balance the branches, e.g., by adding dummy instructions [13] [24], we eliminate these branches.

Algorithm 1 shows our high-level procedure, implemented as an LLVM *function* pass: for each function F, we invoke *BranchMitigationPass(F)* to compute the dominator tree of the control flow graph (CFG) associated with F and then traverse the basic blocks in a depth-first search (DFS) order.

Algorithm 1 Mitigating all sensitive conditional statements.

```
1: function BRANCHMITIGATEPASS(Function F)
2:     let DT(F) be the dominator tree in the CFG of F
3:     for each BasicBlock bb ∈ DT(F) in DFS order do
4:         if bb is the entry of a sensitive conditional statement then
5:             Standardize (bb)
6:             MitigateBranch (bb)
7:         end if
8:     end for
9: end function
```

The dominator tree is a standard data structure in compilers where each basic block has a unique immediate dominator, and an edge from bb_1 to bb_2 exists only if bb_1 is an immediate dominator of bb_2. The DFS traversal order is important because it is guaranteed to visit the inner-most branches before the outer branches. Thus, when *MitigateBranch(bb)* is invoked, we know all branches inside *bb* have been mitigated, i.e., they are either removed or insensitive and hence need no mitigation.

Our mitigation of each conditional statement starting with bb consists of two steps: (1) transforming its IR to a standardized form, using *Standardize(bb)*, to make subsequent processing easier; and (2) eliminating the conditional jumps using *MitigateBranch(bb)*.

3.1.1 Standardizing Conditional Statements

A conditional statement is standardized if it has unique entry and exit blocks. In practice, most conditional statements in cryptographic software are already in standardized. However, occasionally, there may be statements that do not conform to this requirement. For example, in Figure 3.1, the conditional statement inside the while-loop is not yet standardized. In such cases, we transform the LLVM IR to make sure it is standardized, i.e., each conditional statement has a unique entry block and a unique exit block.

Standardization is a series of transformations as illustrated by the examples in Figure 3.2, where

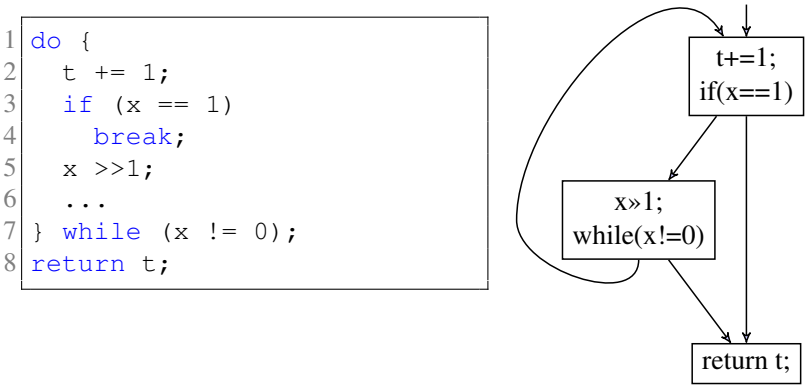


Figure 3.1: A not-yet-standardized conditional statement.

```
for(b=0;b<MAX_B;b++){

    t += 1;
    if (x == 1)
      break;

    x >>1;
    ...
    if (x == 0)
      break;


}
```

```
no_br2 = 1;
for(b=0;b<MAX_B;b++){
  if (no_br2) {
    t += 1;
    if (x == 1)
      break;

    x >>1;
    ...
    if (x == 0)
      no_br2 = 0;

  }
}
```

```
no_br1= no_br2= 1;
for(b=0;b<MAX_B;b++){
  if (no_br1&&no_br2){
    t += 1;
    if (x == 1)
      no_br1 = 0;
    if (no_br1) {
      x >>1;
      ...
      if (x == 0)
        no_br2 = 0;
    }
  }
}
```

Figure 3.2: Standardized conditional statements (Fig. 3.1).

auxiliary variables such as `no_br1` and `no_br2` are added to make the loop bound independent of sensitive variables. `MAX_B` is the bound computed by our conservative static analysis; in cryptographic software, it is often 64, 32, 16 or 8, depending on the number of bits of the integer variable `x`.

Algorithm 2 Mitigating the conditional statement from bb.

function MITIGATEBRANCH(BasicBlock bb)
  Let $cond$ be the branch condition associated with bb
  for each Instruction i in THEN branch or ELSE branch **do**
    if i is a **Store** of the value val to the memory address $addr$ **then**
      Let $val' = \textbf{\textit{CTSEL}}(cond, val, \textbf{\textit{Load}}(addr))$
      Replace i with the new instruction $\textbf{\textit{Store}}(val', addr)$
    end if
  end for
  for each Phi Node ($\%rv \leftarrow \phi(\%rv_T, \%rv_E)$) at the merge point **do**
    Let $val' = \textbf{\textit{CTSEL}}(cond, \%rv_T, \%rv_E)$
    Replace the Phi Node with the new instruction ($\%rv \leftarrow val'$)
  end for
  Change the conditional jump to THEN branch to unconditional jump
  Delete the conditional jump to ELSE branch
  Redirect the outgoing edge of THEN branch to start of ELSE branch
end function

3.1.2 Replacing Conditional Statements

Given a standardized conditional statement, we perform a DFS traversal of its dominator tree to guarantee that we always mitigate the branches before their merge point. The pseudo code, shown in Algorithm 2, takes the entry block bb as input.

Condition and CTSEL: First, we assume the existence of *CTSEL(c,t,e)*, a constant-time intrinsic function that returns t when c equals *true*, and e when c equals *false*. Without any target-specific optimization, it may be implemented using bit-wise operations: CTSEL(c,t,e) {c0=c−1; c1= ~c0; val= (c0 & e)|(c1 & t);} — when the variables are of 'char' type and c is $true$, $c0$ will be $0x00$ and c_1 will be $0xFF$; and when c is $false$, $c0$ will be $0xFF$ and $c1$ will be $0x00$. With target-specific optimization, CTSEL(c,t,e) may be implemented more efficiently. For example, on x86 or ARM CPUs, we may use CMOVCC instructions as follows: {MOV val t; CMP c, 0x0; CMOVEQ val e;} which requires only three instructions. We will demonstrate through experiments (Section 7.4) that target-specific optimization reduces the runtime overhead significantly.

Store Instructions: Next, we transform the branches. If the instruction is a *Store(val,addr)* we replace it with *CTSEL*. That is, the *Store* instructions in `THEN` branch will only take effect when the condition is evaluated to $true$, while the *Store* instructions in `ELSE` branch will only take effect when condition is $false$.

Local Assignments: The above transformation is only for memory *Store*, not assignment to a register variable such as if (cond) {rv=val1; ...} else {rv=val2; ...} because, inside LLVM, the latter is represented in the static single assignment (SSA) format. Since SSA ensures each variable is assigned only once, it is equal to if (cond) {%rv1=val1; ...} else {%rv2=val2; ...} together with a *Phi Node* added to the merge point of these branches.

The Phi Nodes: The Phi nodes are data structures used by compilers to represent all possible values of local (register) variables at the merge point of CFG paths. For $\%rv \leftarrow \phi(\%rv_T, \%rv_E)$, the variables $\%rv_T$ and $\%rv_E$ in SSA format denote the last definitions of $\%rv$ in the THEN and ELSE branches: depending on the condition, $\%rv$ gets either $\%rv_T$ or $\%rv_E$. Therefore, in our procedure, for each Phi node at the merge point, we create an assignment from the newly created val' to $\%rv$, where val' is again computed using `CTSEL`.

Unconditional Jumps: After mitigating both branches and the merge point, we can eliminate the conditional jumps with unconditional jumps. For the standardized branches on the left-hand side of Figure 3.3, the transformed CFG is shown on the right-hand side.

3.1.3 Optimizations

The approach presented so far still has redundancy. For example, given if (cond) {∗addr=∗val_T;} else {∗addr=val_E;} the transformed code would be ∗addr = CTSEL(cond,val_T,∗addr); ∗addr = CTSEL(cond,∗addr,val_E); which has two CTSEL instances. We remove one or both CTSEL instances:

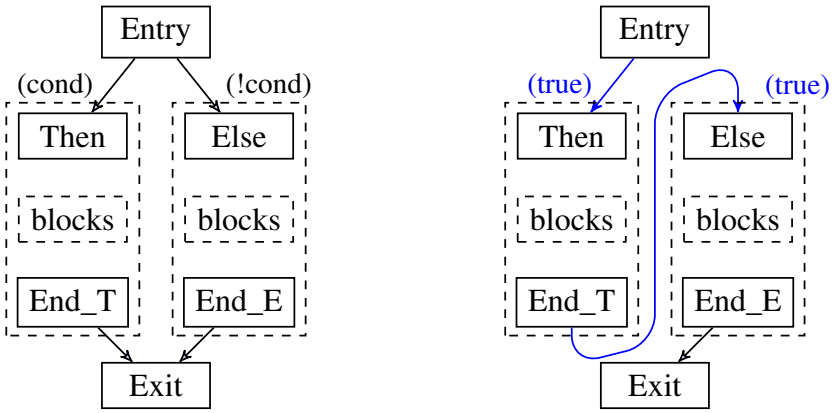


Figure 3.3: Removing the conditional jumps.

- If (val_T==val_E) holds, we merge the two *Store* operations into one *Store*: ∗addr = val_T
- Otherwise, we use ∗addr = CTSEL(cond,val_T,val_E)

In the first case, all CTSEL instances are avoided. Even in the second case, the number of CTSEL instances is reduced by half.

3.2 Mitigating Lookup Table Accesses

We present our method for mitigating lookup table accesses that may lead to cache-timing leaks. In cryptographic software, such leaks are often due to dependencies between indices used to access S-Boxes and the secret data. However, before delving into the details of our method, we perform a theoretical analysis of the runtime overhead of various alternatives, including those designed against the more-capable attackers. Then we review the static cache analysis from [65, 66] before presenting the mitigation.

3.2.1 Mitigation Granularity and Overhead

We focus on the less-capable attackers who only observe the *total execution time* of the victim's program. Under this threat model, we develop optimizations to take advantage of the cache structure and unique characteristics of the software being protected. Our mitigation, illustrated by the example in Figure 3.4, can be significantly more efficient than the approaches illustrated in Figure 2.4.

In contrast, the *Byte-access-aware* threat model allows attackers to observe timing characteristics of each instruction in the victim's program, which means mitigation would have to be applied to every LUT access to make sure there is no timing difference (Figure 2.3).

The *Line-access-aware* threat model allows attackers to see the difference between memory locations mapped to different cache lines. Thus, mitigation only needs to touch all cache lines associated with the table (Figure 2.4).

Let π be a path in P and $\tau(\pi)$ be its execution time. Let τ_{max} be the maximum value of $\tau(\pi)$ for all possible π in P. For our *Total-time-aware* threat model, the ideal mitigation would be a program P' whose execution time along all paths matches τ_{max}. In this case, we say mitigation has *no additional* overhead. We quantify the overhead of other approaches by comparing to τ_{max}.

Table 3.1 summarizes the results. Let N be the table size, CLS be the cache line size, and $M = \lceil N/CLS \rceil$ be the number of cache lines needed. Let K be the number of times table elements are accessed. Without loss of generality, we assume each element occupies one byte. In the best case where all K accesses are mapped to the same cache line, there will be 1 miss followed by $K - 1$ hits. In the worst case (τ_{max}) where the K accesses are scattered in M cache lines, there will be M misses followed by $K - M$ hits.

When mitigating at the granularity of a byte (e.g., Figure 2.3), the total number of accesses in P' is increased from K to $K * N$. Since all elements of the table are touched when any element is read,

Table 3.1: Overhead comparison: N is the table size; $M = \lceil N/CLS \rceil$ is the number of cache lines to store the table; K is the number of times table elements are accessed.

Program Version	# Accesses	# Cache Miss	# Cache Hit
Original program	K	from M to 1	from K-M to K-1
Granularity: Byte-access	K*N	M	K*N-M
Granularity: Line-access	K*M	M	K*M-M
Granularity: Total-time (τ_{max})	K	M	K-M
Our Method: opt. w/ cache analysis	K+M-1	M	K-1

all M cache lines will be accessed. Thus, there are M cache misses followed by $K * N - M$ hits.

When mitigating at the granularity of a line (e.g., Figure 2.4), the total number of accesses becomes $K * M$. Since all cache lines are touched, there are M cache misses followed by $K * M - M$ hits.

Our method, when equipped with static cache analysis based optimization (Section 5.3.2), further reduces the overhead: by checking whether the table, once loaded to the cache, will stay there until all accesses complete. If we can prove the table never gets evicted, we only need to load each line once. Consequently, there will be M misses in the first loop iteration, followed by $K - 1$ hits in the remaining $K - 1$ loop iterations.

In all cases, however, the number of cache misses (M) matches that of the ideal mitigation; the differences is only in the number of cache hits, which increases from $K - M$ to $K * N - M$, $K * M - M$, or $K - 1$. Although these numbers (of hits) may differ significantly, the actual time difference may not, because a cache hit often takes an order of magnitude shorter time than a cache miss.

3.2.2 Static Cache Analysis

We first review a basic static cache analysis, which were previously used in execution time estimation [65, 66]. In our case, it is used to decide whether a memory element is definitely in the cache.

```
1  //mitigation #6: preloading sbox[256] during the first loop
       iteration block_0 = block[0];
2  for (j=block_0 % CLS; j < 256; j+=CLS) {
3    sbox_j = sbox[j];
4    val = (block_0 == j)? sbox_j : block_0;
5  }
6  block[0] = val;
7  //access to sbox[...] is always a hit
8  for (i = 1; i < 16; ++i) {
9    block[i] = sbox[block[i]];
10 }
```

Figure 3.4: Reduction: preloading only in the first iteration.

The Abstract Domain

We design our static analysis procedure based on the unified framework of abstract interpretation [44], which defines a suitable abstraction of the program's state as well as transfer functions of all program statements. There are two reasons for using abstract interpretation. The first one is to ensure the analysis can be performed in finite time even although precise analysis of the program may be undecidable. The second one is to summarize the analysis results along all paths and for all inputs.

Let $V = \{v_1, ..., v_n\}$ be the set of program variables, each of which is mapped to a subset $L_v \subseteq L^*$ of cache lines. The age of $v \in V$, denoted $Age(v)$, is a set of integers corresponding to ages (subscripts) of the lines it may reside (e.g., along all paths and for all inputs). Let the cache be fully associative with the LRU replacement policy, which means a variable $v \in V$ may be mapped to any cache line, and if there is not enough space, the least recently used (LRU) variable will be evicted from the cache. Assume that N is the total number of cache lines, we can define the age $Age(v)$ for each variable $v \in V$, which is an integer ranging from 1 to $N + 1$. Here, $Age(v) = 1$ means v resides in the most recently used line, $Age(v) = N$ means v resides in the least recently used cache line, and $Age(v) = N + 1$ means v is outside of the cache. The program's cache state, denoted $S = \langle Age(v_1), \ldots, Age(v_n) \rangle$, provides the ages of all variables.

Consider an example program with three variables x, y and z, where x is mapped to the first cache line, y may be mapped to the first two lines (e.g., along two paths) and z may be mapped to Lines 3-5. Thus, $L_x = \{l_1\}$, $L_y = \{l_1, l_2\}$, and $L_z = \{l_3, l_4, l_5\}$, and the cache state is $\langle\{1\}, \{1, 2\}, \{3, 4, 5\}\rangle$.

In this context, a *Must-Hit* analysis needs to compute, at each program location, an upper bound of $Age(v)$. If the upper bound is less than or equal to N, then v must be in the cache. Otherwise, it is possible that v may be outside of the cache.

The Transfer Function

Let TRANSFER$(S, inst)$ be the transfer function that models the impact of executing $inst$ in the cache state S: given the current state $S = \langle Age(v_1), \ldots, Age(v_n)\rangle$, it returns a new state $S' = \langle Age'(v_1), \ldots, Age'(v_n)\rangle$. If $inst$ does not access memory at all, then $S' = S$. Otherwise, assume that $v \in V$ is the variable being accessed in $inst$, and we compute the new state S' as follows:

- For the accessed variable v, set $Age'(v) = 1$ in S'.
- For variable $u \in V$ whose age may be younger than v in S, increment the age of u; that is, $Age(u) < Age(v) \rightarrow Age'(u) = Age(u) + 1$.
- For any other variable $w \in V$, set $Age'(w) = Age(w)$.

Given the definition of TRANSFER for an instruction, we define it for a sequence of instructions, denoted $Insts = \{inst_0, inst_1, ...inst_n\}$, as follows: TRANSFER$(S, Insts) =$

$$\text{TRANSFER}(\ldots(\text{TRANSFER}(S, inst_0), inst_1), \ldots, inst_n).$$

Figure 3.5 show two examples. The left-hand-side example illustrates the access of v, which is not

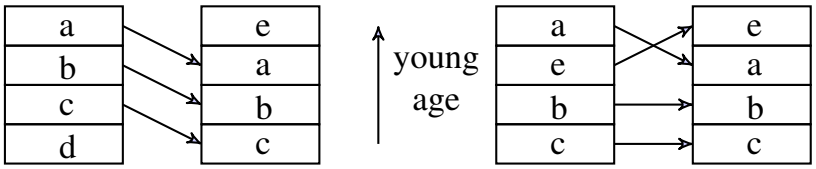


Figure 3.5: Transfer of the cache state under the LRU policy.

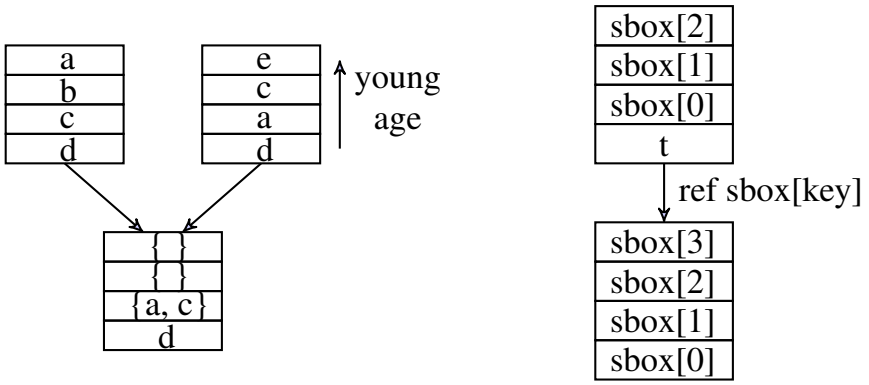


Figure 3.6: Update of the abstract cache state: (1) on the left-hand side, join at the merge point of two paths; and (2) on the right-hand side, a non-deterministic key for memory access.

yet loaded into the cache. After the access, $Age(v) = 1$, meaning v is loaded to the youngest cache line. Furthermore, the ages of all other lines increase by 1. Since $Age(u_4) > 4$, the variable u_4 is evicted from the cache.

In the right-hand-side example, however, v is in the cache prior to the execution of the instruction. Thus, existing cache lines fall into two categories. For the line (u) whose age used to be younger than that of v, the age increases by 1. For the cache lines (w_1 and w_2) whose ages used to be older than that of v, the ages remain the same.

The Join Operator

For efficiency reasons, cache states computed along two program paths are often joined together at the control-flow merge point, to avoid creating an exponential number of states. In the baseline abstract interpretation algorithm, the join operator ($\sqcup$) always maintains a single cache state in the

result, regardless of how many states are joined.

Therefore we define the join ($\sqcup$) operator accordingly; it is needed to merge states S and S' from different paths. It is similar to set intersection — in the resulting $S'' = S \sqcup S'$, each $Age''(v)$ gets the maximum of $Age(v)$ in state S and $Age'(v)$ in state S'. This is because $v \in V$ is definitely in the cache *only if* it is in the cache according to both states, i.e., $Age(v) \leq N$ and $Age'(v) \leq N$.

Consider the left example in Figure 3.6, where the ages of a are 1 and 3 before reaching the merge point, and the ages of c are 3 and 2. After joining the two cache states, the ages of a and c become 3, and the age of d remains 4. The ages of b and e become $\perp$ because, in at least one of the two states, they are outside of the cache.

Formally, given two states $S = \langle Age(v_1), \ldots, Age(v_n)\rangle$ and $S' = \langle Age(v'_1), \ldots, Age(v'_n)\rangle$, we define $S'' = S \sqcup S'$ as follows: $S'' = \langle max(Age(v_1), Age(v'_1)), \ldots, max(Age(v_n), Age(v'_n))\rangle$.

MUST-HIT Analysis.

Since our goal is to decide whether a memory block is definitely in the cache, we compute in $Age(v)$ the upper bound of all possible ages of v, e.g., along all paths and for all inputs. If this upper bound is $\leq N$, we know v must be in the cache.

Now, consider the right-hand-side example in Figure 3.6, where $sbox$ has four elements in total. In the original state, the first three elements are in the cache whereas $sbox[3]$ is outside. After accessing $sbox[key]$, where the value of key cannot be statically determined, we have to assume the worst case. In our MUST-HIT analysis, the worst case means key may be any index ranging from 0 to 3. To be safe, we assume $sbox[key]$ is mapped to the oldest element $sbox[3]$. Thus, the new state has $sbox[3]$ in the first line while the ages of all other elements are decremented.

3.2.3 Static Cache Analysis-based Reduction

As described above, we've developed a MUST-HIT analysis which allows us decide if an LUT access needs to be mitigated. For example, in `subCell()` of `LED_encrypt.c` that accesses `sbox[16]` using `for(i=0; i<4; i++) for(j=0;j<4;j++) { state[i][j]=sbox[state[i][j]];}` since the size of `sbox` is 16 bytes while a cache line has 64 bytes, all the elements can be stored in the same cache line. Therefore, the first loop iteration would have a cache miss while all subsequent fifteen iterations would be hits – there is no cache-timing leak that needs mitigation.

There are many other real-world applications where accesses to lookup tables result in MUST-HITs. For example, block ciphers often consist of multiple encryption or decryption rounds, each of which performs computation using the same lookup table. Instead of mitigating every round, we use our cache analysis to check if, starting from the second round, mitigation can be skipped.

Correctness and Termination.

Our analysis is a conservative approximation of the actual cache behavior. For example, when it says a variable has age 2, its actual age must not be older than 2. Therefore, when it says the variable is in the cache, it is guaranteed to be true, i.e., our analysis is sound; however, it is not (meant to be) complete in finding all MUST-HIT cases – insisting on being both sound and complete could make the problem undecidable. In contrast, by ensuring the abstract domain is finite (with finitely many lines in L and variables in V) and both TFunc and ($\sqcup$) are monotonic, we guarantee that our analysis always terminates.

Handling Loops.

One advantage of using abstract interpretation [44] is the capability of handling loops: for each *back edge* in the CFG, cache states from all incoming edges are merged using the join ($\sqcup$) operator.

Nevertheless, loops in cryptographic software have unique characteristics. For example, most of them have fixed loop bounds, and many are in functions that are invoked in multiple encryption/decryption rounds. Thus, memory accesses often cause cache misses in the first loop iteration of the first function invocation, but hits subsequently. Such *first-misses* followed by *always hits*, however, cannot be directly classified as MUST-HITs.

To exploit such unique characteristics, we perform a code transformation prior to our analysis. We unroll the first iteration out of the loop while keeping the remaining iterations. For example, `for(i=0;i<16;++i) {block[i]=...}` become `{block[0]=...} for(i=1;i<16;++i) {block[i]=...}`. As soon as accesses in the first iteration are mitigated, e.g., as in Figure 3.4, all subsequent loop iterations will result in MUST-HITs, meaning we can skip the mitigation and avoid the runtime overhead. Our experiments on a large number of real applications show that the cache behaviors of many loops can be exploited in this manner.

3.3 Related Work

Kocher [86] is perhaps the first to demonstrate the feasibility of timing side-channel attacks in embedded systems. Since then, timing attacks have been demonstrated on many other platforms [15, 23, 36, 42, 73, 81, 121]. For example, Brumley et al. [36] demonstrated that timing attacks could be carried out remotely through a computer network. Cock et al. [42] found timing side channels in the seL4 microkernel and then performed a quantitative evaluation. Sung et al. [151] used LLVM transformations together with software verification tools to conduct cache timing analysis. Guo et al. [75] demonstrated the existence of concurrency-related cache timing leaks using a technique named adversarial symbolic execution.

Noninterference properties [15, 24, 78, 90, 128] have been formulated to characterize side-channel leaks. To quantify the amount of leaks, Millen [111] proposed to use Shannon's channel capa-

city [144], which models the correlation between sensitive data and timing observations. Other approaches, including min-entropy [145] and g-leakage [18], were also proposed. Backes and Köpf [21] developed an information-theoretic model for quantifying side-channel information. Köpf and Smith [91] proposed a technique for bounding the leakage in blinded cryptographic algorithms.

Prior techniques for removing timing leaks focused primarily on conditional branches, e.g., type-driven cross-copying proposed by Agat [13]. Molnar et al. [114] introduced, along the *program counter (PC)* model, a method for merging branches. Köpf and Mantel [90] proposed a unification-based technique encompassing the previous two methods. Independently, Barthe et al. [24] proposed a transactional branching technique that leverages the availability of commit/abort operations. Coppens et al. [43] developed a compiler backend for removing instruction-timing channels on x86 processors. However, Mantel and Starostin [107] recently compared four of these existing techniques on Java byte-code, and showed none was able to eliminate the leaks completely. Furthermore, these methods did not consider cache-timing leaks.

Our method focuses on leveraging program transformations to completely eliminate timing channels caused by both sensitive conditionals and cache. However, there are also techniques that do not attempt to eliminate timing leaks but hide via randomization or blinding [20, 33, 46, 79, 86, 89, 169]. For example, noise may be added [79] and software diversification may be leveraged to confuse attackers [46]. Correlation between sensitive operations and their observed execution time may be changed unpredictably [86]; this *blinding* technique has been generalized by Köpf and Dr̈muth [89] to allow trade-offs between the performance overhead and security strength. Askarov et al. [20] proposed to delay the output of a black-box system in order to control its timing side channels; Zhang et al. [169] formalized a similar approach in a programming system to support language-based mitigation of timing channels. There are also hardware-based techniques for eliminating timing channels. Broadly speaking, they fall into two categories: resource isolation and timing

obfuscation. Resource isolation [100, 122, 161] may be realized by partitioning hardware resources to two parts (public and private) and then restrict sensitive data/operations to the private partition. However, it requires modifications of the CPU hardware which is not always possible. Timing obfuscation [79, 133, 157] may be achieved by inserting fixed or random delays, or interfering the measurement of the system clock. In addition to being expensive, it does not eliminate timing channels. Oblivious RAM [72, 99, 149] is another technique for removing leakage through the data flows, but requires a substantial amount of on-chip memory and incurs significant overhead in the execution time. In contrast to all these existing techniques, our method does not require hardware modifications and thus is cheaper and more widely applicable.

Beyond timing side channels, there are techniques for mitigating leaks through other side channels such as power [87, 106] and faults [28]. Some of these techniques have been automated in compiler-like tools [14, 27, 115] whereas others leverage SMT solver-based formal analysis and verification [57, 58, 59, 60, 69, 160, 170] and inductive synthesis techniques [54, 55, 56, 61, 159]. However, none of these techniques was applied to timing side channels.

Chapter 4

Speculative Cache Analysis

In the previous two chapters, we have presented our techniques for detecting and mitigating side-channel leaks, where a static cache analysis is used to help identifying potential leaks and thus driving the mitigation step. The contribution of this chapter is to demonstrate, under micro-architectural optimations, the unsoundness of existing cache analysis techniques, including the one presented in the previous chapters. Then, we present our method for making the cache analysis sound again under speculative execution.

Toward this end, we propose a generally applicable *abstract interpretation* framework for conducting static analysis while maintaining the soundness under speculative execution. This is accomplished by including all possible speculative execution traces. It acheives a good balance between performance and accuracy by carefully choosing the merge points and dynamically changing the speculative boundaries.

It is not tied to any particular way the abstract state is defined, or the abstract domain used to represent the abstract state. For example, the abstract state may be used to capture side effects on the cache content, the pipeline [140, 141], or other CPU components including relaxed memory

models [93, 172]; the non-functional properties may be related to timing or power or other physical characteristics. The abstract domain may be interval, box, or polyhedral domain. In all these above cases, our method is able to capture the speculative execution behaviors missed by existing methods. Although in this work, we have implemented the method to model cache timing behaviors, the analysis framwork can be used to model other nonfunctional properties as well.

4.1 Introduction to Speculative Execution

Speculative execution [153] is a feature that has been implemented by many modern processors. It allows a processor to increase the execution speed by exploring certain program paths ahead of time instead of waiting for the path conditions to be satisfied. This is to prevent slower instructions, e.g., memory accesses, from blocking faster instructions. For example, when a program reaches a branching instruction, e.g., `if(x>5){...}else{...}` where the condition depends on an uncached value of `x` stored in memory, a *non-speculative* execution will force the processor to wait, often for tens or hundreds of clock cycles, until `x` is loaded from memory, whereas *speculative* execution allows the processor to make a prediction of the branching target and then proceed to execute the predicted branch. During speculative execution, the processor maintains a checkpoint of the CPU's register state, which will be used to roll back the changes if the prediction turns out to be incorrect, i.e., after the value of `x` is fetched from memory. However, if the prediction turns out to be correct, speculative execution will save time and thus outperform non-speculative execution.

Speculative execution is designed to be *transparent* to the program running on the processor; that is, it does not affect the program semantics, as the rollback ensures that functional properties are preserved. This is the reason why, in the past, static analysis techniques do not model speculative execution.

However, recent vulnerabilities such as Meltdown [98], Spectre [85] and ForeShadow [156] force

the community to take another look because, although speculative execution preserves the CPU's register state, for performance reasons, it does not preserve the states of many other components such as the cache and the pipeline [70, 71].

4.2 Cache Analysis under Speculative Execution

In modern CPUs, the execution of an instruction takes only 1-3 clock cycles when there is a cache hit, but tens or even hundreds of clock cycles when there is a cache miss. Therefore, static cache analysis is important for analyzing timing related properties of a program, e.g., to detect information leaks through the timing side channel (variances in the execution time of the program), as what we do in Section 3.2. Besides of that, cache analysis can also be used to determine if the execution of a real-time task can finsih before its deadline(known as WCET calculation). For timing side channel detection, in particular, one may want to know if the program's execution time (i.e., the number of cache hits and misses) is not affected by the secret data (e.g., a cryptographic key, security token, or password). For deadline estimation, one may want to compute the number of cache misses along all paths since it directly affect the program's execution time in the worst case. In both applications, the analysis must be *sound* to be useful in practice. By sound, we mean all possible behaviors of the program must be considered during the analysis.

Unfortunately, existing static cache analysis techniques [65, 143] are often unsound since they do not model speculative execution and the possible side effects caused by speculative execution. Although there is a large body of work on improving the accuracy of such analysis [80], in the past, efforts were spent primarily on refining the over-approximations of normal program executions. In the presence of speculative execution, however, the main problem is not caused by too many infeasible program paths being included in the analysis (so one has to design better algorithms to eliminate them), but caused by some real program behaviors being omitted. Therefore, to make the

analysis sound again, we need to develop a different set of ideas from what have been proposed in the aforementioned literature. Specifically, we need to increase (instead of decrease) the set of program behaviors considered in the analysis.

4.2.1 Execution Time Estimation

```
1 char ph[64*510], l1[64], l2[64], p;
2 reg char k;
3 for(reg int i=0;i<64*510; i+=64)
4     load ph[i];
5 if(p==0)
6   load l1[0];
7 else
8   load l2[0];
9 load ph[k];
```

Figure 4.1: Example program for timing side channel.

Figure 4.1 shows a program that illustrates divergent cache behaviors under normal and speculative executions as observed in practice [8, 9, 10, 39, 76, 77]. Here, we have four variables: *ph*, *l1*, *l2*, and *p*, which are mapped to different cache lines. Suppose the register value *k* is 0, the load at line 8 will access *ph[0]*. We assume the cache has 512 lines in total and 64 bytes per line. We also assume the cache is fully associative, meaning any variable may be mapped to a different line. The place holder variable *ph* is mapped to the first 510 lines (line 3); in practice, *ph* may correspond to an assorted set of program variables. Each of the remaining variables, *l1*, *l2* and *p*, may be mapped to a cache line. Depending on the branching condition, either *l1* or *l2* may be loaded to the cache, but both will result in 512 cache misses. As shown on the left-hand side of Figure 4.2, the statement at line 8, accessing *ph[0]*, is always a hit because the content is already in the cache.

However, under speculative execution, upon reaching the *if-else* statement, the CPU needs to load *p* from memory. Due to a cache miss, it performs a speculative execution of the branch `(p==0)`. If the branch prediction is incorrect and the CPU has to roll back the speculative execution, there will

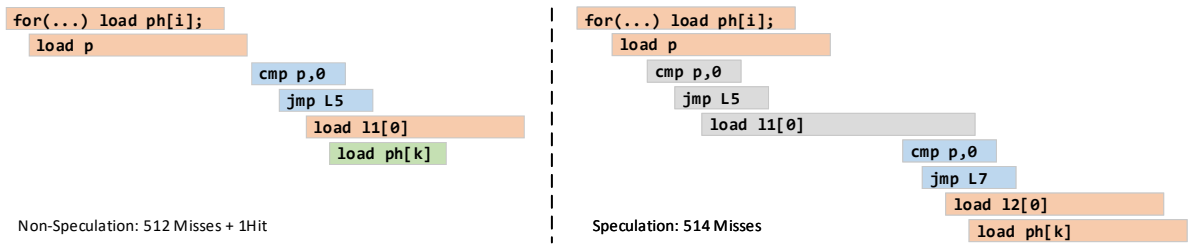


Figure 4.2: Pipelined execution trace for program in Figure 4.1

be 514 cache misses (among which 513 cache misses are observable from outside of the CPU) as shown by the right-hand-side trace in Figure 4.2.

In this case, the program first speculatively executes the *then*-branch and loads `l1` into the cache, and then rolls back to take the *else*-branch and loads `l2`. Although the functional side-effects of executing the *then*-branch are eliminated by the rollback mechanism, `l1` is already in the cache. Since the cache has only 512 lines, following the LRU replacement policy, the first line associated with *ph[0]* is evicted. This is why the subsequent access to *ph[0]* will be a cache miss.

For execution time estimation, the non-speculative execution will lead to 512 cache misses plus 1 cache hit, whereas the speculative execution will lead to 513 observable cache misses (and a speculative cache miss masked by the pipeline). The additional cache miss is important because it will cause a significant delay in the execution time. The message from this example is as follows: if a static analysis is not sound in modeling speculative execution, it may underestimate the worst-case execution time and produce a bogus proof that the computation task meets its deadline.

4.2.2 Side Channel Detection

We use Figure 4.1 again to illustrate a timing side channel made possible by speculative execution. That is, the attacker, by measuring the execution time of a program, may deduce information of the secret data. This time, we assume the variable k stores the secret data, e.g., a cryptographic key, and

the value of k is used as an index to access an *S-Box*-like array named *ph*. If the time taken by the access varies with respect to k, there is an information leak.

In a non-speculative execution, there cannot be leaks in Figure 4.1 because, for all paths and values of *k*, the number of cache misses remains the same. In particular, accessing *ph[k]* is leak-free because the array is loaded to cache at line 3, and executing either branch at lines 5 and 7 will not evict it. However, similar to what we have observed in the execution time estimation example, speculative execution may execute one of the two branches first, and then roll back to execute the other branch. Since the memory locations associated with both branches must be accessed, which add up to more than 512 cache lines, some of the cache lines associated with *ph* will be evicted. Therefore, the subsequent *load* (at line 8) may be a cache miss. The difference in execution time may be observed by the attacker and used to deduce information of the secret k: whether the last statement leads to a cache miss depends on the value of *k*.

4.3 Technical Challenges

The above two examples illustrate the need to soundly model speculative execution. However, there are several challenges. The first one is to model the cache state of a program during speculative execution without drastically altering the abstract interpretation algorithm. The second challenge is to judiciously merge abstract states computed from normal and speculative executions, since *when* and *how* to merge them drastically affect the accuracy of the fixed-point computation. Furthermore, since a speculative execution may be rolled back at any moment, the number of scenarios is exponential in the number of speculatively executed instructions. If we have to enumerate, the analysis time will be prohibitively long. Therefore, we group scenarios into equivalence classes, based on which we perform reduction to balance the performance and accuracy.

4.4 Preliminaries

4.4.1 Abstract Interpretation

Abstract interpretation [44] is a static analysis framework that considers all paths and inputs to obtain a sound over-approximation of the state at every program location [92, 93, 150]. For efficiency reasons, the state is kept *abstract* and often represented by a set of constraints in a certain *abstract domain*. For example, in the interval domain, each constraint is of the form $lb \leq x \leq ub$, where x is a variable and lb,ub are the lower and upper bounds. The join of two states, $s_1 = lb_1 \leq x \leq ub_1$ and $s_2 = lb_2 \leq x \leq ub_2$, is defined as $s_1 \sqcup s_2 = min(lb_1, lb_2) \leq x \leq max(ub_1, ub_2)$. Here, $\sqcup$ denotes the *join* operator, which returns an over-approximation of the set union. If, for example, the polyhedral abstract domain is used, a constraint will be a linear equation and the *join* operator may be the convex hull.

The purpose of restricting the representation of states to an abstract domain is to reduce the computational overhead. Although various abstract domains may be plugged in, the underlying fixed-point computation remains the same. The fixed-point of states are computed on the program's control flow graph (CFG). Without loss of generality, we assume the CFG has a unique entry node and a unique exit node. Inside the CFG, nodes are associated with instructions or basic blocks of instructions, whereas edges represent the control flows, guarded by conditional expressions.

Let TRANSFER : $S \times INST \rightarrow S$ be the transfer function, which takes a state $s \in S$ and an instruction $inst \in INST$ as input, and returns the new state $s' = \text{TRANSFER}(s, inst)$ as output. s' is the result of executing $inst$ in state s.

Algorithm 3 shows a generic procedure that returns, for each CFG node n, an abstract state $S[n]$ as output. $S[n]$ is supposed to be a sound over-approximation of all the possible states at n, regardless of the input values or paths taken to reach n. Initially, $S[n]$ is $\top$ (tautology) for the entry node but $\bot$

Algorithm 3 Abstract interpretation based static analysis.

```
1: Initialize S[n] to ⊤ if n = ENTRY(CFG), and to ⊥ otherwise
2: WL ← ENTRY(CFG)
3: while ∃ n ∈ WL do
4:     WL ← WL \ {n}
5:     s' ← TRANSFER(S[n], inst_n)
6:     for all n' ∈ SUCCESSORS(CFG, n) do
7:         if s' ⋢ S[n']) then
8:             S[n'] ← s[n'] ⊔ s'
9:             WL ← WL ∪ {n'}
10:        end if
11:    end for
12: end while
```

(empty) for all other CFG nodes. The remaining part of the procedure is a standard worklist-based algorithm for computing the fixed point [118]: starting from the entry node, it computes the states of the successor nodes (n') based on the transfer function. To ensure convergence, e.g., when the program has loops or is otherwise non-terminating, a *widening* operator (∇) is needed in addition to *join* ($\sqcup$). However, for brevity, we omit the details; for a complete introduction, refer to [44, 112].

The actual definitions of abstract state S and transfer function TRANSFER depend on the application. In this work, we are concerned with the cache state corresponding to a program.

4.4.2 Cache and Speculative Execution

Cache is a type of small but fast storage to hold frequently used data so that they do not need to be fetched from or stored to the large but slow memory every time. Although this work focuses on the data cache, which is more relevant to our applications, the underlying technique can be extended to the instruction cache as well.

In a typical CPU, e.g., an Intel processor [7], instructions are fetched from memory and decoded continuously before they are sent to the scheduler for execution. Executing an instruction involves multiple units; speculative execution [153] is an optimization that efficiently utilizes these execution

units. During speculative execution, instructions are scheduled in a pipeline as soon as the required execution units are available; for example, while an instruction is waiting for data to be fetched from memory, subsequent instructions may be executed, as long as the program semantics remains the same to observers from outside of the CPU.

Things become complicated when there are branches, however, since the branch prediction unit must make a guess on which branch target to execute. Instructions in the predicted branch will be executed while the branch condition is being evaluated, and will be committed only after the prediction is confirmed to be correct. Upon misprediction, however, the result of speculative execution will be discarded and the execution will be redirected to the correct branch.

The reorder buffer inside the execution unit, among others, is responsible for this *rollback*: upon a branch mis-prediction, it will not perform register retiring as in a normal execution; instead, it will flush out the affected registers, before restoring the CPU to a previously saved state.

The branch predictor also plays an important role in speculative execution since its accuracy is directly related to the performance of the CPU. However, regardless of the underlying strategies [82, 158, 167], when a branch prediction turns out to be incorrect, the speculatively executed instructions may leave side-effects on the states of other system components, including the cache. In this work, we are concerned with modeling of such side-effects in abstract interpretation.

4.5 Modeling the Speculative Execution

In this section, we lift the baseline abstract interpretation algorithm so that it can soundly model speculative execution.

4.5.1 Augmented CFG with Virtual Control Flow

Given the CFG of a program, we first augment it by adding special nodes and edges, to model all possible control flows produced by speculative executions. These implicit control flows, which will be made explicit in our augmented CFG, are called the *virtual control flows*.

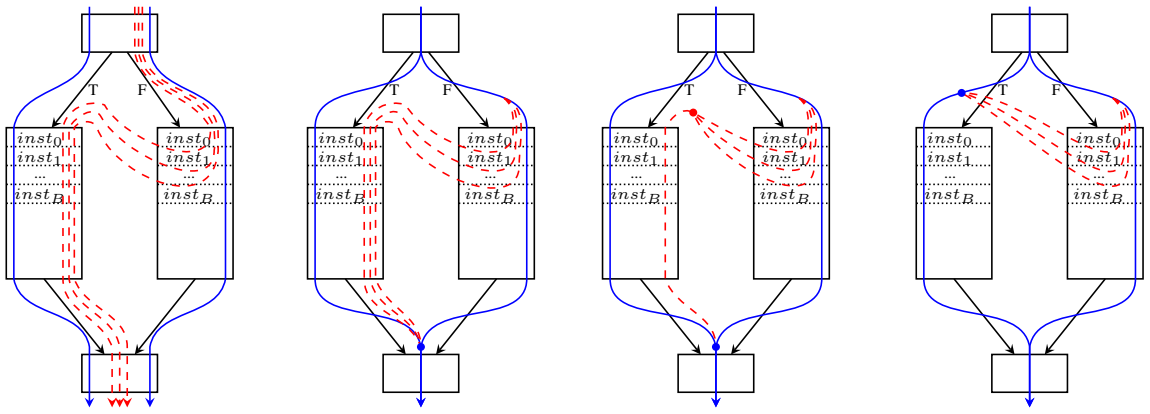
A virtual control flow occurs at every *if-else* statement where the branching condition depends on some variables stored in memory. In a normal execution, a branch guarded by a condition (c) is explored only when c is satisfied. However, under speculative execution, the branch will be explored (speculatively) by our algorithm even if c is unsatisfiable. Furthermore, upon mis-prediction, the rollback will re-direct the control to the other branch.

To model all these behaviors, we add the following special nodes and edges to the CFG for every branch that may be explored speculatively:

- vn_{start}, which is a special CFG node that denotes the start of a virtual control flow;
- vn_{stop}, which is a special CFG node that denotes the end of a virtual control flow.

The edges connecting such nodes, which represent the virtual control flows, fall into five categories: (1) n–vn_{start}; (2) vn_{start}–n; (3) n–n; (4) n–vn_{stop}, and (5) vn_{stop}–n, where n is a normal CFG node.

The edge n–vn_{start} represents the start of a speculative execution: it feeds the state $S[n]$ to vn_{start}, which in turn generates a speculative state $SS[vn_{start}] = S[n]$. Then, the newly created speculative state is propagated through the edge vn_{start}–n. Next, it is propagated through the edges n–n and n–vn_{stop} until reaching vn_{stop}–n. The special node vn_{stop} converts the speculative state $SS[vn_{stop}]$ back to the normal state $S[n] = SS[vn_{stop}]$. Afterward, the state is joined with other states from the normal execution.



(a) flows without merging (b) merged after branch (c) merged before branch (d) merged into normal flow

Figure 4.3: Strategies for merging speculative control flows.

One way to add the special nodes and edges is illustrated in Figure 4.3a. Specifically, for each *if-else* statement, we add virtual control flow edges from instructions in one of the branch to the entry node of the other branch under the same branching condition.

Here, the blue solid lines represent normal executions, whereas the red dashed lines represent virtual control flows associated with speculative executions of the *else*-branch. Virtual control flows associated with the *then*-branch are similar, but omitted in the figure for clarity. The reason why there are more than one dashed lines is because the roll-back point (i.e., location where roll-back occurs) is non-deterministic; to be conservative, we assume it may occur at any moment within the maximum speculation depth.

In practice, the *speculation depth* is platform-dependent and bounded by a few factors [68, 127], e.g., the size of the reorder buffer; the maximum number of unresolved branches that the CPU can handle before it stalls; whether there are division-by-zero or floating-point errors in the program; and the number of clock cycles taken to access memory and resolve a branching condition. For simplicity, for example, we assume that the maximum speculative execution depth is provided by the user. In Figure 4.3a, we assume that $inst_B$ is the boundary within which roll-back occurs.

4.5.2 Merging the Speculative Flows

Since we use abstract interpretation to over-approximate the cache states, multiple executions must be merged to reduce the computational overhead. In the baseline algorithm, for example, states from two different paths are joined whenever the program paths are merged in the CFG. In the speculative analysis, we also need to decide when to join the normal and the speculative states.

Figure 4.3 shows three merging strategies in addition to the original *no-merging* strategy in Figure 4.3a. Consider Figure 4.3b, for example, since the executions before the branch entry node are identical, they are merged without losing accuracy; in addition, the speculative executions are merged right before the exit point of the other branch. Recall that the join operator ($\sqcup$) used to handle merging is over-approximated, we know that the strategy outlined in Figure 4.3b is a sound over-approximation of Figure 4.3a.

To over-approximate even more, consider Figure 4.3c, which merges all speculative states of the *else*-branch before reaching the *then*-branch. However, the merged speculative state is propagated through the *then*-branch before it is merged with the normal state. In contrast, Figure 4.3d is a more aggressive over-approximation, which merges the speculative states with the non-speculative state at the entry node of the *then*-branch.

Regardless of the merging strategy, however, our method ensures that the result is a sound over-approximation.

Since every time state merging occurs, it may lose information, in general, the later that merging occurs, the more accurate the result is, but there is no guarantee. Furthermore, late merging may lead to a more expensive analysis. Our experimental comparisons of these four strategies show that the one outlined in Figure 4.3c is the best: it not only obtains significantly more accurate results than the one in Figure 4.3d, but also runs almost equally fast. Therefore, we have settled down on this strategy: we call it *Just-in-Time* merging.

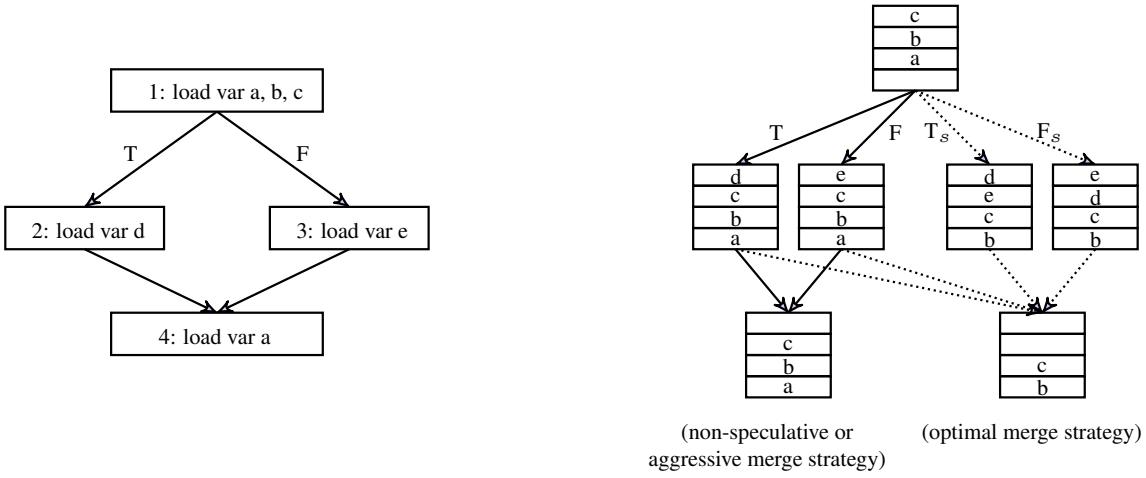


Figure 4.4: Cache state with different merge points.

4.5.3 Just-in-Time Merging: An Example

Consider the CFG of a branch shown on the left-hand side of Figure 4.4, where each basic block refers to a variable (from a to e). The initial cache state, at the top of the figure on the right-hand side, is the state after executing the first basic block, where variables a, b and c are loaded into the cache. Here, the solid arrows represent the normal execution, where either d or e is mapped to the youngest cache line. Since we are concerned with a *Must-Hit* analysis, after merging at basic block 4, only a, b and c are left in the cache.

Under speculative execution, we may execute the *else*-branch before rolling back to execute the *then*-branch. If we choose to merge the speculative state right after the rollback, the merging would be between d, c, b and a on the one hand, and e, d, c and b on the other hand. The merged state will not contain e anymore, thus losing the important information of speculative execution.

However, if we propagate the speculative state computed from the *else*-branch through the *then*-branch and then merge with the non-speculative state, the cache state at basic block 4 will be more accurate. As shown by the dotted arrow T_s, variable e is loaded to the cache before d is loaded to the cache; similarly, for F_s, variable d is loaded before e is loaded. Finally, when the four states are

merged, the result is that only c and b are guaranteed to result in cache hits. Thus, the cache state on the bottom-right of Figure 4.4, which corresponds to *Just-in-Time* merging illustrated in Figure 4.3c, captures the side effect of speculative execution.

4.6 Generalization and Optimization

In this section, we present the generalized algorithm before discussing several optimizations, which help increase accuracy as well as decrease runtime overhead.

Algorithm 4 shows the static analysis procedure that is sound under speculative execution. Given the original CFG of a program, it first constructs an augmented CFG by adding the virtual control flows. Then, it initializes the abstract states for each program location n, including both the default state, denoted $S[n]$, and the speculative state, denoted $SS[n]$. Next, it starts the fixed-point computation using a worklist based procedure that is similar to that of Algorithm 3.

Algorithm 4 Abstract interpretation under speculation.

1: $VCFG \leftarrow$ AUGMENTCFGWITHVIRTUALCONTROLFLOW(CFG)
2: Initialize $S[n]$ to $\top$ if $n \in$ ENTRY($VCFG$), else to $\bot$
3: Initialize $SS[n]$ to $\bot$ for all $n \in VCFG$
4: $WL \leftarrow$ ENTRY($VCFG$)
5: **while** $\exists n \in WL$ **do**
6: $\quad WL \leftarrow WL \setminus \{n\}$
7: $\quad$ **if** n is a normal CFG node **then**
8: $\quad\quad s' \leftarrow$ TRANSFER($S[n]$, n)
9: $\quad\quad ss' \leftarrow$ TRANSFER($SS[n]$, n)
10: $\quad$ **else**
11: $\quad\quad$ Set ss' to $S[n]$ if n is a special n_{start} node, else to $\bot$
12: $\quad\quad$ Set s' to $SS[n]$ if n is a special n_{stop} node, else to $\bot$
13: $\quad$ **end if**
14: $\quad$ **for each** $n' \in$ SUCCESSORS($VCFG$, n) **do**
15: $\quad\quad$ **if** $s' \not\sqsubseteq S[n']$ or $ss' \not\sqsubseteq SS[n']$ **then**
16: $\quad\quad\quad S[n'] \leftarrow S[n'] \sqcup s'$
17: $\quad\quad\quad SS[n'] \leftarrow SS(n') \sqcup ss'$
18: $\quad\quad\quad WL \leftarrow WL \cup \{n'\}$
19: $\quad\quad$ **end if**
20: $\quad$ **end for**
21: **end while**

However, when the special CFG node vn_{start} is encountered (Line 11), the default state $S[n]$, which is from the incoming edge, is used to create a speculative state $ss' \leftarrow S[n]$; this is to model the side effects caused by the failed speculative execution upon rollback. From then on, both the default state $S[n]$ and the speculative state $SS[n]$ will be propagated through subsequent nodes in the VCFG; at each node n, the transfer function has to be applied to both of them (Lines 8-9). This continues until the other special node vn_{stop} is encountered, which transforms the speculative state $SS[n]$ back to s' (Line 12) before s' is merged into the normal flow.

4.6.1 The Running Example

To illustrate how the algorithm works, consider the example program in Figure 4.5, which is a real-time DSP program written in C [76]. The corresponding CFG is shown in Figure 4.6, where the red (solid and dashed) edges represent the two virtual control flows.

Result from Non-speculative Executions Table 4.1 shows the cache state computed for each location (basic block) based on only normal executions (black edges in Figure 4.6); this is analogous to running the baseline procedure in Algorithm 3. In Column 2, the variables are arranged according to their ages: the younger variable appears on the left.

Initially, the cache is empty. From basic block 1 to 5, we apply the transfer functions: `decis_lev` takes two cache lines, but since we do not unwind the loop, we do not know its index statically. Thus, we nondeterministically pick one for the first time, `decis_lev[1*]`. Following the back edge from basic block 4, when `decis_lev` is accessed again, we conservatively choose the second cache line for `decis_lev[2*]` to ensure that the cache state remains an over-approximation. Our analysis iterates through the loop three times before it reaches a fixed-point (light gray row) and terminates.

```
/* table is 31-byte long to make quantl look-up
easier, last entry is for mil=30 when wd is max */
int quant26bt_pos[31] = { 61,60,59,58,57,56,55,54,
  53,52,51,50,49,48,47,46,45,44,43,42,41,40,39,
  38,37,36,35,34,33,32,32 };
/* table is 31-byte long to make quantl look-up
easier, last entry is for mil=30 when wd is max */
int quant26bt_neg[31] = { 63,62,31,30,29,28,27,26,
  25,24,23,22,21,20,19,18,17,16,15,14,13,12,11,10,
  9,8,7,6,5,4,4 };
/* decision levels - pre-multiplied by 8 */
int decis_levl[30] = { 280,576,880,1200,1520,1864,
  2208,2584,2960,3376,3784,4240,4696,5200,5712,
  6288,6864,7520,8184,8968,9752,10712,11664,12896,
  14120,15840,17560,20456,23352,32767 };

int quantl(int el,int detl) {
  int ril,mil;
  long int wd,decis;
  /* abs of difference signal */
  wd = my_abs(el);
  /* mil based on decision levels and detl gain */
  for(mil = 0 ; mil < 30 ; mil++) {
    decis = (decis_levl[mil]*(long)detl) >> 15L;
    if(wd <= decis) break;
  }
  /*if mil=30, wd is less than all decision levels*/
  if(el >= 0) ril = quant26bt_pos[mil];
  else ril = quant26bt_neg[mil];
  return(ril);
}
```

Figure 4.5: Code snippet from a real-time DSP program [76].

Result from Speculative Executions Table 4.2 shows the cache state computed under speculative execution. For clarity, we only focus on the cache states relevant to the speculative executions starting from basic block 5. We use two different colors, blue and red, to differentiate the cache states computed from non-speculative (blue) and speculative (red) executions. By considering speculative executions, it is possible for us to access both `quant26bt_pos` and `quant26bt_neg` in a single execution.

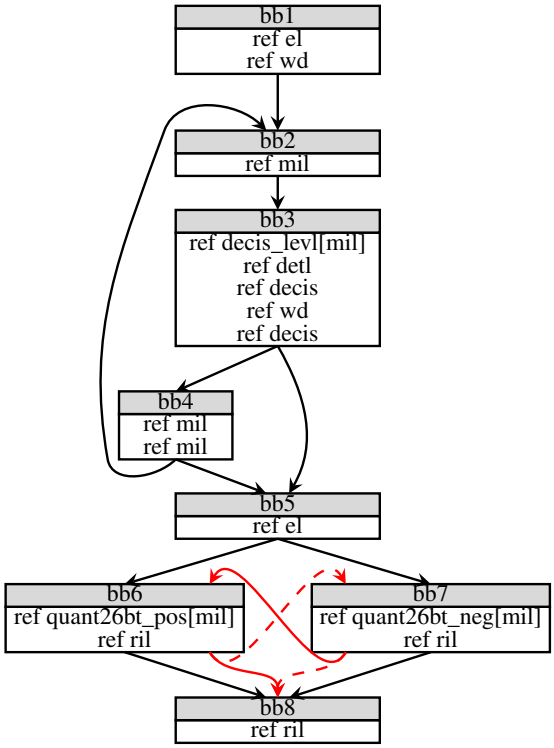


Figure 4.6: Augmented CFG with virtual control flows.

Table 4.1: Cache states during the fixed-point computation.

BBlk	Cache State
0	{ }
1	{wd, el}
2	{mil, wd, el}
3	{decis, wd, detl, decis_lev[1*], mil, el}
4	{mil,decis, wd, detl, decis_lev[1*], el}
2	{mil,decis, wd, detl, decis_lev[1*], el}
3	{decis, wd,detl, decis_lev[2*], mil, decis_lev[1*], el}
4	{mil, decis, wd,detl, decis_lev[2*], decis_lev[1*], el}
2	{mil, decis, wd,detl, decis_lev[2*], decis_lev[1*], el}
5	{el ,decis, wd,detl, decis_lev[2*], mil, decis_lev[1*]}
6	{ril, quant26bt_pos[1*], el ,decis, wd,detl, decis_lev[2*], mil, decis_lev[1*]}
7	{ril, quant26bt_neg[1*], el ,decis, wd,detl, decis_lev[2*], mil, decis_lev[1*]}
8	{ril, $\emptyset$, el ,decis, wd,detl, decis_lev[2*], mil, decis_lev[1*]}

Execution Time Estimation The last row of Table 4.2, which differs from the last row of Table 4.1, shows that most of the program variables have older ages than before. This is dangerous

Table 4.2: Cache states during speculative execution.

BBlk	Cache State
...	...
5	{el ,decis, wd,detl, decis_lev[2*], mil, decis_lev[1*]}
6	{ril, quant26bt_pos[1*], el ,decis, wd,detl, decis_lev[2*], mil, decis_lev[1*]}
7	{ril, quant26bt_neg[1*], el ,decis, wd,detl, decis_lev[2*], mil, decis_lev[1*]}
6	{ril, quant26bt_pos[1*], el ,decis, wd,detl, decis_lev[2*], mil, decis_lev[1*]}
7	{ril, quant26bt_neg[1*], el ,decis, wd,detl, decis_lev[2*], mil, decis_lev[1*]}
7	{ril, quant26bt_neg[1*], quant26bt_pos[1*], el ,decis, wd,detl, decis_lev[2*], mil, decis_lev[1*]}
6	{ril, quant26bt_pos[1*], quant26bt_neg[1*], el ,decis, wd,detl, decis_lev[2*], mil, decis_lev[1*]}
8	{ril, $\emptyset$, el ,decis, wd,detl, decis_lev[2*], mil, decis_lev[1*]}
8	{ril, $\emptyset$, $\emptyset$, el ,decis, wd,detl, decis_lev[2*], mil, decis_lev[1*]}

```
1  #define BUF_SIZE 1024*16
2  const uint8_t sbox[256] = { 0x63,0x7c,0x77,0x7b,0xf2,0x6b,0x6f,0xc5,
3          0x30,0x01,0x67,0x2b,0xfe,0xd7,0xab,0x76, ...};
4  int main()
5  {
6    uint32_t inBuf[BUF_SIZE];
7    int el, delt, tmp;
8    for(int i=0; i< 256; i++) // preload sbox
9    tmp = sbox[i];
10   for(int i=0; i< BUF_SIZE; i++) // read inBuf
11   tmp = inBuf[i];
12   tmp = quantl(el, delt);
13   AES_encode(inBuf);
14 }
```

Figure 4.7: The client code that leads to side-channel leaks.

because, if the cache is only large enough to hold the first eight variables, there will be an additional cache miss, which may force the program to miss its deadline.

Side Channel Detection The additional cache miss may also lead to side-channel leaks. Figure 4.7 shows a client program that uses the program in Figure 4.5. The application first accepts some input from the user, then processes it using *quantl* as a subroutine, and finally encrypts the result using a cipher such as AES. Before calling *quantl*, a look-up table named *sbox* is loaded; the lookup table will be used by the cipher while it encrypts the data, during which time a secret *key* is used as the index to access *sbox*.

By controlling the input size, a malicious user can force part of the *sbox* to be evicted from the cache. As a result, for some *key* values, accessing *sbox* results in a cache hit, but for other *key* values, it results in a cache miss. Although timing side channels have been investigated before [25, 51, 75, 165], these prior works never considered speculative execution. Our contribution, in this context, is to show that even if a program is *leak-free* under normal execution, it may still be leaky under speculative execution.

4.6.2 Dynamically Bounding Speculation Depth

Although the maximum number of speculatively executed instructions is used to construct the augmented CFG, in practice, the number of speculatively executed instructions can be smaller. For example, when all variables needed to resolve a branching condition are in the cache, speculative execution may be shortened. Since our cache analysis aims to decide whether a variable access is a *must-hit*, as the analysis continues it may report more *must-hit* variables, which can be used to bound the speculation depths of other branches.

Thus, we propose an optimization that leverages the *must-hit* variables to dynamically remove virtual control flows that are deemed redundant. Toward this end, we maintain two predefined bounds for each speculative execution, b_h and b_m, which correspond to the branching condition being a cache hit or miss. (Since b_h and b_m are platform-dependent, they are set based on input from the user.) By default, we use b_m as the bound; but as soon as the branching condition is proved to be a must-hit, we switch the bound to b_h.

This optimization not only decreases the computational overhead, i.e., by reducing the number of edges in the VCFG, but also increases the accuracy since it results in a potentially tighter over-approximation. In the extreme case where $b_h = 0$, for example, switching to b_h means avoiding speculative execution all together, which can avoid many bogus behaviors.

While our focus here is on exploiting changes to the speculation depth due to cache misses, the proposed technique may be extended to exploit other sources of changes, e.g., execution units being busy, or division taking a longer time based on the operands.

4.6.3 Handling the Merges and Loops

The algorithm presented so far uses the join operator ($\sqcup$) to over-approximate the union of two abstract states. However, in the presence of loops, it may have limitations: (1) the resulting state may not be accurate enough, and (2) it may take a long time (or forever) to reach a fixed point.

Thus, we add a widening operator [45] to the standard join operation $s[n'] \sqcup s'$; that is, we use $(s[n'] \sqcup s')\nabla s'$ instead of $s[n'] \sqcup s'$. The idea behind widening (∇) is simple: first, we identify the *direction of growth* from the state s' to the state $(s[n'] \sqcup s')$; then, we over-approximate $(s[n'] \sqcup s')$ in such a way that it maximizes the progress along the *direction of growth*. In the interval domain, for example, if the previous state is $s' = 0 \leq x \leq 3$ and the current state is $s = 0 \leq x \leq 5$, the result of widening would be $s \nabla s' = 0 \leq x \leq +\infty$. To achieve better accuracy, loops with fixed iteration number will be fully unrolled; only unresolved loops will be widened.

Figure 4.8 shows another loop-related problem. First, variable `a` is loaded into the cache. Then, inside the loop, every time the branch is executed, $Age(a)$ increases by 1. After the join, however, neither `b` nor `c` will be in the cache. Thus, eventually, a is evicted from the cache as well. This is not accurate because, during the actual execution, a will never be evicted. With a refined join operator, we will be able to avoid this problem.

We refine the join operator ($\sqcup$) by adding extra information into the cache state. Similar to Touzeau et al. [154], we introduce a shadow variable $\exists v$ for each $v \in V$. Whenever two states are merged and v appears in only one of the two states, the shadow variable $\exists v$ will remain in the merged cache (while the normal variable v will not). Figure 4.9 shows an example, where both `b` and `e` will be

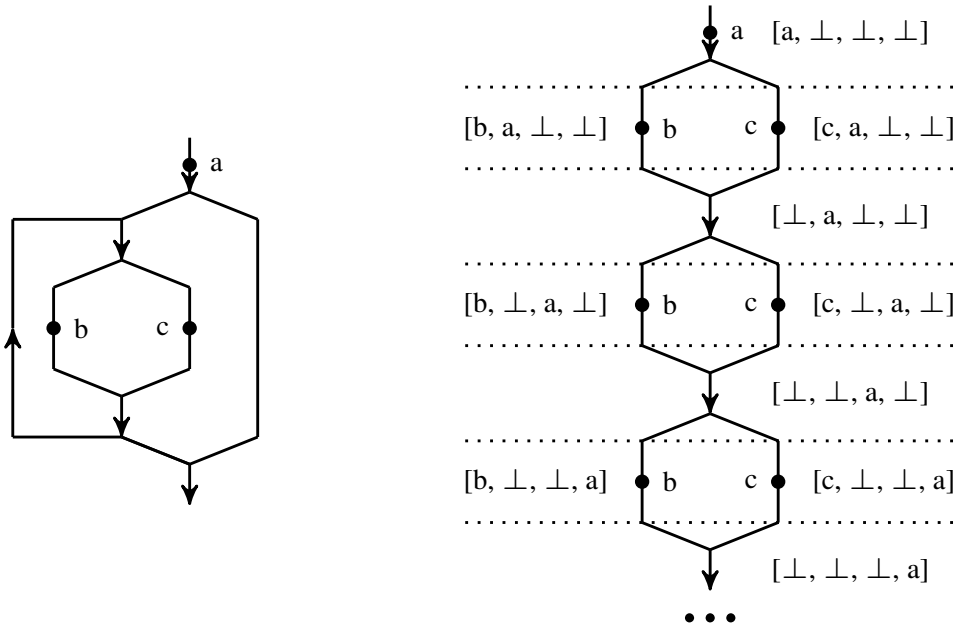


Figure 4.8: Example program for the widening operator.

replaced with $\exists b$ and $\exists e$ in the final cache state. That is, there exists a path in which variable b or c is cached.

We also revise the transfer function: the shadow variable $\exists v$ will be removed if a concrete reference to v is applied. For example, in Figure 4.9, if the variable b is accessed on the merged state, $\exists b$ will be removed from final state.

For simplicity, we unroll the loop for three times and illustrate the sequence of memory accesses in Figure 4.10. The abstract cache states are listed on both sides at each memory access and merge point. With the shadow variables, our cache states are able to reach the fixed-points after only three iterations and avoid evicting a.

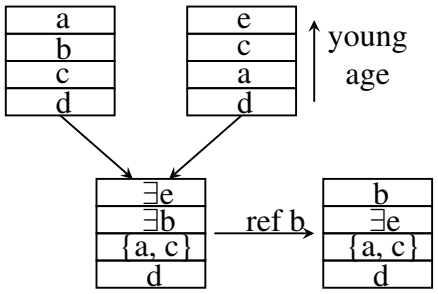


Figure 4.9: Transfer function with shadow variables.

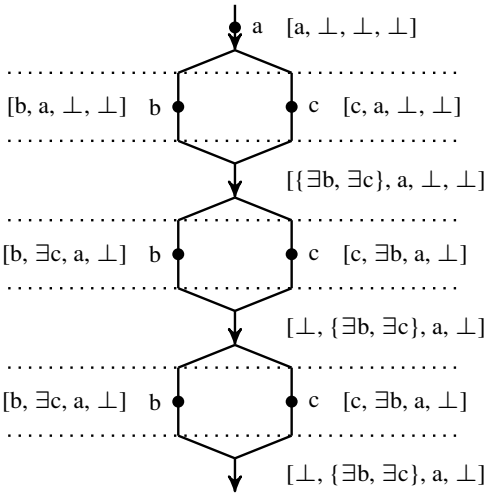


Figure 4.10: The refined join using shadow variables.

4.6.4 Handling Multiple Speculative Executions

Finally, we extend our algorithm so it can independently propagate the speculative states through the virtual control flows, without interfering each other, even if one branching statement is embedded inside another branching statement.

Algorithm 5 shows the procedure, which computes, for each node n in the augmented CFG, a set of states of the form $SS[n][c]$, one for each speculative execution. Let $C = \{1, \ldots, k\}$ be the set of all branches in the program that can be speculatively executed; each $1 \leq i \leq k$ is the index of a branch in this set. We call $c = i$ the color of the i-th speculative execution. While constructing the VCFG, for each $c \in C$, we add a separate set of virtual control-flow edges and nodes, with the color c.

During the fixed-point computation, instead of applying the transfer function once to generate a speculative state ss', the procedure applies the transfer function $|C|$ times, to generate a vector of speculative states $ss'[c]$, one for each speculative execution with color c. As such, every speculative execution (of color $c \in C$) is handled separately until the corresponding node n_{stop} (of the same color c) is encountered, in which case the speculative state $SS[n][c]$ is transformed back to a

Algorithm 5 Analysis under a set of speculative executions.

```
1: (VCFG, C) ← AUGMENTCFGWITHVIRTUALCONTROLFLOW(CFG)
2: Initialize S[n] to ⊤ if n ∈ ENTRY(VCFG), else to ⊥
3: Initialize SS[n][c] to ⊥ for all n ∈ VCFG and for all color c ∈ C
4: WL ← ENTRY(VCFG)
5: while ∃n ∈ WL do
6:     WL ← WL \ {n}
7:     if n is a normal CFG node then
8:         s' ← TRANSFER( S[n], n)
9:         ss'[c] ← TRANSFER( SS[n][c], n) for all color c ∈ C
10:    else
11:        Set s' to SS[n][c] if n is node n_start of color c, else to ⊥
12:        Set ss'[c] to S[n] if n is node n_stop of color c, else to ⊥
13:    end if
14:    for each n' ∈ SUCCESSORS(VCFG, n) do
15:        if s' ⋢ S[n'] or ∃c : ss'[c] ⋢ SS[n'][c] then
16:            S[n'] ← S[n'] ⊔ s'
17:            SS[n'][c] ← SS(n') ⊔ ss'[c] for all color c ∈ C
18:            WL ← WL ∪ {n'}
19:        end if
20:    end for
21: end while
```

non-speculative state s'.

There are alternative ways of presenting the analysis procedure in Algorithm 5, for example, by using the trace partitioning framework developed by Mauborgne and Rival [108]. Also note that, for ease of comprehension, we choose to split the speculative states from the normal states. However, the two types of states may be treated uniformly and processed using a generalized worklist-based algorithm. Assume that the worklist-based algorithm is smart enough, the special merge nodes created for virtual control flows can be viewed as merely optimization hints.

4.7 Related Work

Abstract interpretation [44] is a framework for conducting static analysis and proving properties. Ferdinand and Wilhelm [66, 67] pioneered the use of abstract interpretation in may- and must-hit cache analyses [162]. Others also used similar techniques to detect timing side channels [25,

51, 165]. However, prior works focused primarily on improving abstract interpretation without considering speculative execution.

There are some techniques that consider the impact of speculative execution [95], but only for the instruction pipeline. In a commercial tool named *AIT*, speculations are also considered during execution time estimation by leveraging a standalone pipeline analysis as a driver [162]. Since the tool is propriety, details of this analysis have not been made public; therefore, it is not clear how speculative execution is modeled during abstract interpretation.

Our method differs from the large body of work on statistically estimating the worst-case execution time of real-time software [94, 96, 113] using either CPU simulators or characteristics of prior simulation results [152]. These techniques, while useful, are not designed to be sound, and hence may not be suitable for the applications that we have in mind, such as detecting side-channel leaks or proving that leaks do not exist. The reason is because, if the analysis is not sound, the proof may not be valid and as a consequence, leaks may be left undetected.

For timing side channels, many analysis and verification techniques [23, 35, 37, 75, 124, 126, 151, 165] have been developed. Some of these methods, however, only consider instruction-related timing variance while ignoring the cache completely. They include, for example, the method developed by Chen et al. [38], which uses Cartesian Hoare Logic [147] to prove that timing leaks of a program are bounded, the method developed by Antonopoulos et al. [19], which uses a similar technique for proving the absence of timing channels, and the method developed by Nilizadeh et al. [119], which uses differential fuzzing to show the existence of timing leaks.

There are also techniques for improving the accuracy of cache analysis, e.g., by using symbolic execution or model checking to refine the cache analysis results [40, 110, 154] and by extending the analysis from single-core to multi-core CPUs [104]. However, none of these techniques considered speculative execution, which is the focus of our work.

Chapter 5

Runtime Enforcement under Burst Error

In the previous two chapters, we have presented techniques for statically mitigating side-channel leaks to improve the security of systems. In this chapter, we present our methods for enforcing safety of reactive systems at run time. Here, the enforcers are synthesized from the safety properties automatically.

5.1 Introduction to the Shield

First, we introduce the technical background and define the terminologies used in this chapter.

5.1.1 The Reactive System

A reactive system is a system that continuously responds to external events. In practice, reactive systems may have strict timing requirements that demand them to respond without any delay. Furthermore, they are often safety-critical in that a violation may lead to catastrophe. In this context, it is important to guarantee the certainty that the system satisfies a small set of safety properties even

in the presence of design defect and environmental disturbance. However, traditional verification and fault-tolerance techniques cannot accomplish this task. In particular, fault-tolerance techniques are not effective in dealing with design defects whereas verification techniques are not effective in dealing with transient faults introduced by the environment. Furthermore, formal verification techniques such as model checking are limited in handling large designs and third-party IP cores without the source code.

The reactive system to be protected by the shield is represented as a Mealy machine $\mathcal{D} = \langle S, s_0, \Sigma_I, \Sigma_O, \delta, \lambda \rangle$, where S is a finite set of states, $s_0 \in S$ is the initial state, Σ_I is the set of values of the input signals, Σ_O is the set of values of the output signals, δ is the transition function, and λ is the output function. More specifically, $\delta(s, \sigma_I)$ returns the unique next state $s' \in S$ for a given state $s \in S$ and a given input value $\sigma_I \in \Sigma_I$, while $\lambda(s, \sigma_I)$ returns the unique output value $\sigma_O \in \Sigma_O$.

The safety specification that we want to enforce is represented as a finite automaton $\varphi^s = \langle Q, q_0, \Sigma, \delta_\varphi, F_\varphi \rangle$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, $\Sigma = \Sigma_I \times \Sigma_O$ is the input alphabet, δ_φ is the transition function, and $F_\varphi \subseteq Q$ is a set of unsafe (error) states. Let $\overline{\sigma} = \sigma_0 \sigma_1 \ldots$ be an input trace where for all $i = 0, 1, \ldots$ we have $\sigma_i \in \Sigma$. Let $\overline{q} = q_0 q_1 \ldots$ be the corresponding state sequence such that, for all $i = 0, 1, \ldots$, we have $q_{i+1} = \delta_\varphi(q_i, \sigma_i)$.

We assume the input trace $\overline{\sigma}$ of φ^s is generated by the reactive system $\mathcal{D}$. We say that $\overline{\sigma}$ satisfies φ^s if and only if the corresponding state sequence $\overline{q}$ visits only the safe states; that is, for all $i = 0, 1, \ldots$ we have $q_i \in (Q \setminus F_\varphi)$. We say that $\mathcal{D}$ satisfies φ^s if and only if all input traces generated by $\mathcal{D}$ satisfies φ^s. Let $L(\varphi^s)$ be the set of all input traces satisfying φ^s. Let $L(\mathcal{D})$ be the set of all input traces generated by $\mathcal{D}$. Then, $\mathcal{D}$ satisfies φ^s if and only if $L(\mathcal{D}) \subseteq L(\varphi^s)$.

5.1.2 The Safety Shield

Following Bloem et al. [31], we define the shield as another reactive system $\mathcal{S}$ such that, even if $\mathcal{D}$ violates φ^s, the combined system $(\mathcal{D} \circ \mathcal{S})$ still satisfies φ^s. We define the synchronous composition of $\mathcal{D}$ and $\mathcal{S}$ as follows:

Let the shield be $\mathcal{S} = \langle S', s'_0, \Sigma, \Sigma_{O'}, \delta', \lambda' \rangle$, where S' is a finite set of states, $s'_0 \in S'$ is the initial state, $\Sigma = \Sigma_I \times \Sigma_O$ is the input alphabet, $\Sigma_{O'}$, which is the set of values of O', is the output alphabet, $\delta' : S' \times \Sigma \to S'$ is the transition function, and $\lambda' : S' \times \Sigma \to \Sigma_{O'}$ is the output function.

The composition is $\mathcal{D} \circ \mathcal{S} = \langle S'', s''_0, \Sigma_I, \Sigma_{O'}, \delta'', \lambda'' \rangle$, where $S'' = (S \times S')$, $s''_0 = (s_0, s'_0)$, Σ_I is the set of values of the input of $\mathcal{D}$, $\Sigma_{O'}$ is the set of values of the output of $\mathcal{S}$, δ'' is the transition function, and λ'' is the output function. Specifically, $\lambda''((s, s'), \sigma_I)$ is defined as $\lambda'(s', \sigma_I \cdot \lambda(s, \sigma_I))$, which first applies $\lambda(s, \sigma_I)$ to compute the output of $\mathcal{D}$ and then uses $\sigma_I \cdot \lambda(s, \sigma_I)$ as the new input to compute the final output of $\mathcal{S}$. Similarly, δ'' is a combined application of δ and λ from $\mathcal{D}$ and δ' from $\mathcal{S}$. That is, $\delta''((s, s'), \sigma_I) = (\delta(s, \sigma_I), \delta'(s', \sigma_I \cdot \lambda(s, \sigma_I)))$.

Let $L(\mathcal{D} \circ \mathcal{S})$ be the set of input traces generated by the composed system. Clearly, if $L(\mathcal{D}) \subseteq L(\varphi^s)$, the shield $\mathcal{S}$ should simply maintain ${\sigma_O}' = \sigma_O$. But if $L(\mathcal{D}) \not\subseteq L(\varphi^s)$, the shield $\mathcal{S}$ needs to modify the original output of $\mathcal{D}$ to eliminate the erroneous behaviors in $L(\mathcal{D}) \setminus L(\varphi^s)$.

In general, there are multiple ways for $\mathcal{S}$ to change the original output $\sigma_O \in \Sigma_O$ into ${\sigma_O}' \in \Sigma_{O'}$ to eliminate the erroneous behaviors, some of which are better than others in minimizing the deviation. Ideally, we would like the shield to do nothing when $\mathcal{D}$ satisfies φ^s; that is, ${\sigma_O}' = \sigma_O$. However, when $\mathcal{D}$ violates φ^s, the deviation is inevitable. In this case, the shield synthesized by Bloem et al. [31] guarantees that the deviation is minimum only if there are no multiple errors within each k-step recovery period. Under burst error, however, their shield would enter a fail-safe mode where it stops minimizing the deviation. This is undesirable because, even after the transient errors disappear, their shield would still keep modifying the output values.

Our synthesis method takes a safety specification φ^s of the reactive system $\mathcal{D}(I, O)$ as input, and returns another reactive system $\mathcal{S}(I, O, O')$ as output. Following Bloem et al. [31], we call $\mathcal{S}$ a shield. We use I and O to denote the set of input and output signals of the original system, respectively, and define the runtime enforcer $\mathcal{S}(I, O, O')$ as follows: It takes I and O as input and returns a modified version of O as output to guarantee the combined system satisfies the safety specification; that is, $\varphi^s(I, O')$ holds even if $\varphi^s(I, O)$ is violated. Furthermore, the shield modifies O only when $\varphi^s(I, O)$ is violated, and even in that case, it tries to minimize the deviation between O and O'. This approach has several advantages. First, since $\mathcal{S}$ is a reactive system, it can correct the erroneous output in O in the same clock cycle. Second, since $\mathcal{S}$ is agnostic to the size and complexity of the system $\mathcal{D}$, it is cheaper and more scalable than fault-tolerance techniques. Finally, the approach works even if the design contains third-party IP cores.

5.1.3 Example of Shield Handling Burst Error

Now we use an example to illustrate the main advantage of our shield synthesis method, which is the capability of handling burst error. Consider the automaton representation of a safety specification in Fig. 5.1, which has three states, one Boolean input signal, and two Boolean output signals. Here, the state 0 is the initial state and the state 2 is the unsafe state. Every edge in the figure represents a state transition. The edge label represents the values of the input and output signals, where the digit before the comma is for the input signal and the two digits after the comma are for the output signals. X stands for *don't care*, meaning that the digit can be either true (1) or false (0). Among other things, the safety specification in Fig. 5.1 states that when the input value is 0, the two output values cannot be 11; furthermore, in state 1, the two output values cannot be 00.

Assume that the design $\mathcal{D}(i, o_1o_2)$ occasionally violates the safety specification, e.g, by generating 11 for the output signals o_1o_2 when the input i is 0, which forces the automaton to enter the unsafe

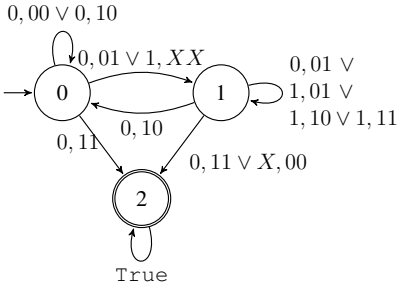


Figure 5.1: Example safety specification φ^s.

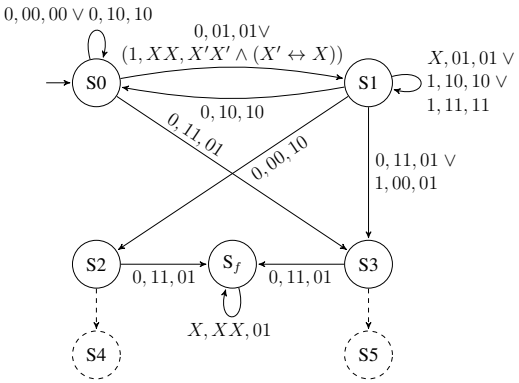


Figure 5.2: The 2-stabilizing shield [31].

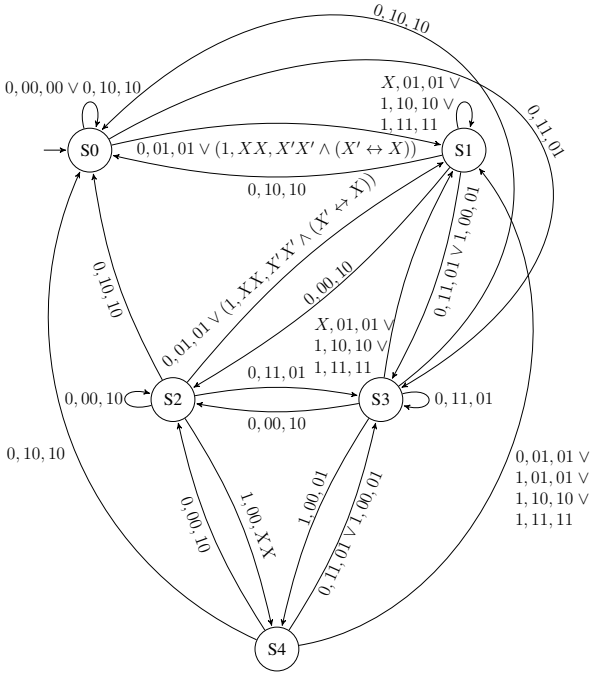


Figure 5.3: Our new shield for burst error.

state. We would like to have the shield $\mathcal{S}(i, o_1o_2, o_1'o_2')$ to produce correct values for the modified output $o_1'o_2'$ as either 10, 01, or 00. Furthermore, whenever the design satisfies the specification or recovers from transient errors, we would like to have the shield produce the same (correct) output as the design; that is, $o_1' = o_1$ and $o_2' = o_2$.

Unfortunately, the shield synthesized by Bloem et al. [31] can not always accomplish this task. Indeed, if given the safety specification in Fig. 5.1 as input, their method would report that a 1-stabilizing shield, which is capable of recovering from a violation in one clock cycle, does not exist, and the best shield their method can synthesize is a 2-stabilizing shield, shown in Fig. 5.2 (to make it simple, we omit part of the shield unrelated to handle burst error), which requires up to 2 clock cycles to fully recover from a property violation. For example, starting from the initial state

Step	0	1	2	3	4	5	6	7	8	9
Input i	0	0	1	0	0	0	0	0	0	...
Output o_1o_2	00	01	10	11	11	10	10	00	00	...
Shield output $o_1'o_2'$	00	01	10	01	01	01	01	01	01	...
State in Fig. 5.2	S0	S0	S1	S1	S3	S_f	S_f	S_f	S_f	...

Figure 5.4: Simulation trace of 2-stabilizing shield.

Step	0	1	2	3	4	5	6	7	8	9
Input i	0	0	1	0	0	0	0	0	0	...
Design Output o_1o_2	00	01	10	11	11	10	10	00	00	...
Shield output $o_1'o_2'$	00	01	10	01	01	10	10	00	00	...
State in Fig. 5.3	S0	S0	S1	S1	S3	S3	S0	S0	S0	...

Figure 5.5: Simulation trace of our new shield.

$S0$, if the shield sees $i, o_1o_2 = 0, 01$, which satisfies φ^s, it will produce $o_1'o_2' = 01$ and go to the state $S1$. From $S1$, if the shield sees $i, o_1o_2 = 0, 11$, which violates φ^s, it will produce $o_1'o_2' = 01$ and go to the state $S3$. At this moment, if the second violation $i, o_1o_2 = 0, 11$ occurs, the shield will enter a *fail-safe* state S_f, where it stops minimizing the deviation between $o_1'o_2'$ and o_1o_2.

Fig. 5.4 shows the simulation trace where two consecutive errors occur in Steps 3 and 4, forcing the shield to enter the fail-safe state s_f where it no longer responds to the original output o_1o_2. This is shown in Steps 5-8, where the original output no longer violates φ^s and yet the shield still modifies the values to 01.

In contrast, our new method would synthesize the shield shown in Fig. 5.3, which never enters any fail-safe state but instead keeps minimizing the deviation between $o_1'o_2'$ and o_1o_2 even in the presence of burst error. As shown in the simulation trace in Fig. 5.5, when the two consecutive violations occur in Steps 3 and 4, our new shield will correct the output values to 01. Furthermore, immediately after the design recovers from the transient errors, the shield stops modifying the original output values. Therefore, in Steps 5-8, our shield maintains $o_1'o_2' = o_1o_2$.

5.2 Synthesize Shield under Burst Error

In this section, we present our new shield synthesis algorithm for handling burst error.

5.2.1 The Overall Flow

Algorithm 6 shows the overall flow of our synthesis procedure. The input of the procedure consists of the safety specification $\varphi^s(I, O)$, and the set of signals in I, O, and O'. The output of the procedure is the safety shield $\mathcal{S}(I, O, O')$.

Algorithm 6 Synthesizing the shield $\mathcal{S}(I, O, O')$ from the safety specification $\varphi^s(I, O)$.

```
1: SYNTHESIZE (specification φ^s, input I, output O, modified output O') {
2:   Q(I, O') ← GENCORRECTNESSMONITOR(φ^s)
3:   E(I, O, O') ← GENERRORAVOIDINGMONITOR(φ^s)
4:   G ← Q ∘ E        // create the safety game
5:   ρ ← COMPUTEWINNINGSTREGETY(G)
6:   S(I, O, O') ← CONSTRUCTSHIELD(ρ)
7:   return S
8: }
```

Starting from the safety specification φ^s, our synthesis procedure first constructs a correctness monitor $\mathcal{Q}(I, O')$. The correctness monitor $\mathcal{Q}$ ensures that the composed system, whose input is I and output is O', always satisfies the safety specification. That is, $\varphi^s(I, O')$ holds even if $\varphi^s(I, O)$ occasionally fails. Note that $\mathcal{Q}(I, O')$ alone may not be sufficient as a specification for synthesizing the desired shield $\mathcal{S}$, because it refers only to O' but not to O. For example, if we give $\mathcal{Q}$ to a classic reactive synthesis procedure, e.g., Pnueli and Rosner [129], it may produce a shield that ignores the original output O of the design and arbitrarily generates O' to satisfy $\varphi^s(I, O')$.

To minimize the deviation from O to O', we construct an error-avoiding monitor $\mathcal{E}(I, O, O')$ from φ^s. In this work, we use the Hamming distance between O and O' as the measurement of the deviation. Therefore, when the design $\mathcal{D}(I, O)$ satisfies $\varphi^s(I, O)$, the error-avoiding monitor ensure that $O' = O$. When $\mathcal{D}(I, O)$ violates $\varphi^s(I, O)$, however, we have to modify the output to avoid the violation of $\varphi^s(I, O')$; in such cases, we want to impose constraints in $\mathcal{E}$ so as to minimize the deviation from O to O'. The detailed algorithm for constructing $\mathcal{E}$ is presented in Section 5.2.2. Essentially, $\mathcal{E}(I, O, O')$ captures all possible ways of modifying O to O' to minimize the deviation.

To pick the best possible modification strategy, we formulate the synthesis problem as a two-player safety game, where the shield corresponds to a winning strategy. Toward this end, we define a set of *unsafe* states of $\mathcal{E}$ as follows: they are the states where $\varphi^s(I, O)$ holds but $O' \neq O$, and they must be avoided by the shield while it modifies O to O'.

The two-player safety game is played in the game graph $\mathcal{G} = \mathcal{Q} \circ \mathcal{E}$, which is a synchronous composition of the correctness monitor $\mathcal{Q}$ and the error-avoiding monitor $\mathcal{E}$. Recall that $\mathcal{Q}$ is used to make sure that $\varphi^s(I, O')$ holds, and $\mathcal{E}$ is used to make sure that $O' = O$ whenever $\varphi^s(I, O)$ holds. Therefore, the set of *unsafe* states of $\mathcal{G}$ is defined as follows: they are the states that are unsafe in either $\mathcal{Q}$ or $\mathcal{E}$. Conversely, the *safe* states of $\mathcal{G}$ are those that simultaneously guarantee $\varphi^s(I, O')$ and minimum deviation from O to O'. The main difference between our new synthesis method and the method of Bloem et al. [31] is in the construction of this safety game: their method does not allow the second error to occur in O during the k-step recovery period of the first error, whereas our new method allows such error.

After solving the two-player safety game denoted as $\mathcal{G}(I, O, O')$, we obtain a winning strategy $\rho = (\delta_\rho, \lambda_\rho)$, which allows us to stay in the safe states of $\mathcal{G}$ by choosing proper values of O' regardless of the values of I and O. The winning strategy consists of two parts: δ_ρ is the transition function that takes a present state of $\mathcal{G}$ and values of I and O as input and returns a new state of $\mathcal{G}$, and λ_ρ is the output function that takes a present state of $\mathcal{G}$ and values of I and O as input and returns a new value for O'. Finally, we convert the winning strategy ρ into the shield $\mathcal{S}$, which is a reactive system that implements the transition function and output function in ρ.

5.2.2 Constructing the Safety Game

We first use an example to illustrate the construction of the safety game $\mathcal{G}$ from φ^s. Consider Fig. 5.6 (a), which shows the automaton representation of a safety property of the ARM bus

Figure 5.6: Example: (a) safety specification $\varphi^s(R, S)$ and (b) correctness monitor $\mathcal{Q}(R, S')$.

arbiter [30]; the LTL formula is $\mathsf{G}(\neg R \rightarrow \mathsf{X}(\neg S))$, meaning that transmission cannot be *started* (S is the output) if the bus is not *ready* (R is the input signal). In Fig. 5.6 (a), the state 2 is unsafe. The first step of our synthesis procedure is to construct the correctness monitor $\mathcal{Q}(R, S')$, shown in Fig. 5.6 (b), which is a duplication of $\varphi^s(R, S)$ except for replacing the original output S with the modified output S'.

The next step is to construct the error-avoiding monitor $\mathcal{E}(R, S, S')$, which captures all possible ways of modifying S into S' to avoid reaching the unsafe state. This is where our method differs from Bloem et al. [31] the most. Specifically, Bloem et al. [31] assume that the second violation from the design will not occur during the k-step recovery period of the first violation. If there are more than one violations within k steps, it would enter a *fail-safe* state S_f, where it stops tracking the deviation from S to S'. Our method, in contrast, never enters the *fail-safe* state. It starts from the safety specification φ^s and replaces all transitions to the *unsafe* state with transitions to some safe states. This is achieved by modifying the value of the output signal S so that the transition matches some existing transition to a safe state. If there are multiple ways of modifying S to redirect the edges leading to unsafe states in φ^s, we simultaneously track all of these choices until the ambiguity is completely resolved. In other words, we keep correcting consecutive violations without ever giving up (entering S_f). This is done by modifying the error tracking automaton which is responsible for motoring the behavior of design: we conservatively assume the design will make mistakes at any time, so whenever there is a chance for the design to make mistakes, we generate a new abstract state to guess its correct behaviors.

Construction of $\mathcal{E}(I, O, O')$

Algorithm 7 shows the pseudocode for constructing the error-avoiding monitor $\mathcal{E}$. At the high level, $\mathcal{E} = \mathcal{U} \circ \mathcal{T}$, where $\mathcal{U}(I, O)$ is called the violation monitor and $\mathcal{T}(O, O')$ is called the deviation monitor.

- To construct the violation monitor $\mathcal{U}$, we start with a copy of the specification automaton φ^k, and then replace each existing edge to a failing state, denoted as $(s, l) \rightarrow t$, with an edge to a newly added abstract state s_g, denoted as $(s, l) \rightarrow s_g$. The abstract state s_g represents the set of possible safe states to which we may redirect the erroneous edge. That is, each safe state $s' \in s_g.states$ may be reached from s through $(s, l') \rightarrow t'$, where l, l' share common input label. Since each guessing state s_g represents a subset of the safe states in φ^s, the procedure for constructing $\mathcal{U}(I, O)$ from $\varphi^s(I, O)$ resembles the classic procedure for subset construction.

- To construct the deviation monitor $\mathcal{T}$, we start by creating two states A and B and treating values of O and O' as the input symbols. Whenever $O = O'$, the state transition goes to state A, and whenever $O \neq O'$, the state transition goes to B. Finally, we label A as the safe state and B as the unsafe state. Fig. 5.9 shows the deviation monitor.

Consider the safety specification $\varphi^s(R, S)$ in Fig. 5.6 (a) again. To construct the violation monitor $\mathcal{U}(R, S)$, we first make a copy of the automaton φ^s, as shown in Line 2 of Algorithm 7. Then, starting from Line 3, we replace the edge to the unsafe state 2, denoted as $(1, S) \rightarrow 2$, with the edge to a guessing state, denoted as $(1, S) \rightarrow 2_g$, where the set of safe states in 2_g is $\{0, 1\}$. That is, if we modify the output value S to the new value $\neg S$, the transition from state 1 may go to either state 0 or state 1. This is shown in Fig. 5.7 (a). In Lines 6-8, for each outgoing edge of the states in $\{0, 1\}$, we add an outgoing edge from 2_g.

Algorithm 7 Generating error-avoiding monitor $\mathcal{E}$ from safety specification φ^s.

```
1:  GENERRORAVOIDINGMONITOR ( specification φ^s ) {
2:    U ← copy of the specification automaton φ^s
3:    while (∃ edge (s, l) → t in U where t is an unsafe state) {
4:      Delete edge (s, l) → t from U
5:      Add abstract state s_g and edge (s, l) → s_g into U    //{t'} ⊆ s_g.states
6:      foreach (edge (s, l') → t' such that t' is safe, and l, l' share common input)
7:        foreach (outgoing edge (t', l'') → t'')
8:          Add edge (s_g, l'') → t'' into U
9:      U ← MERGEEDGESWITHSAMELABEL(U)
10:   }
11:   T ← the deviation monitor
12:   E ← U ∘ T
13:   return E
14: }
15: MERGEEDGESWITHSAMELABEL(monitor U) {
16:   while (∃ edges (s_g, l_1) → t_1 and (s_g, l_2) → t_2 in U where l_1 ∧ l_2 is not false) {
17:     Delete edges (s_g, l_1) → t_1 and (s_g, l_2) → t_2 from U
18:     if (l_1 ∧ ¬l_2 is not false)    Add edge (s_g, l_1 ∧ ¬l_2) → t_1 back to U
19:     if (l_2 ∧ ¬l_1 is not false)    Add edge (s_g, l_2 ∧ ¬l_1) → t_2 back to U
20:     Add abstract state s_m and edge (s_g, l_1 ∧ l_2) → s_m to U    //{t_1, t_2} ⊆ s_m.states
21:     foreach (outgoing edge of t_1 and t_2, denoted as (t_12, l') → t')
22:       Add edge (s_m, l') → t' into U
23:     if (t_1 or t_2 is unsafe)    return U
24:   }
25: }
```

Next, we merge the outgoing edges with the same label in Line 9. This acts like a subset construction. For example we may first merge two edges with the label $R \wedge \neg S$, both of them lead to state 0. Then, we merge the two edges with the label $\neg R \wedge \neg S$. Then, consider the edge label $\neg R \wedge S$: starting from state $0 \in 2_g$, the next state is 1, and starting from state $1 \in 2_g$, the next state is 2. Therefore, the outgoing edge labeled $\neg R \wedge S$ goes to the abstract state 4_m, whose set of states is $\{1, 2\}$. Since 2 is an unsafe state, we return back to Line 3 in Algorithm 7 and replace it with other guessing states. More specifically, the state 2 is replaced with the state 1 and 4_m becomes 4_g. After adding all outgoing edges of 4_g, the resulting $\mathcal{U}$ is shown in Fig. 5.7 (a). Similarly, we merge the remaining outgoing edges of 2_g that are labeled $R \wedge S$ and create the abstract state 3_m, whose set of

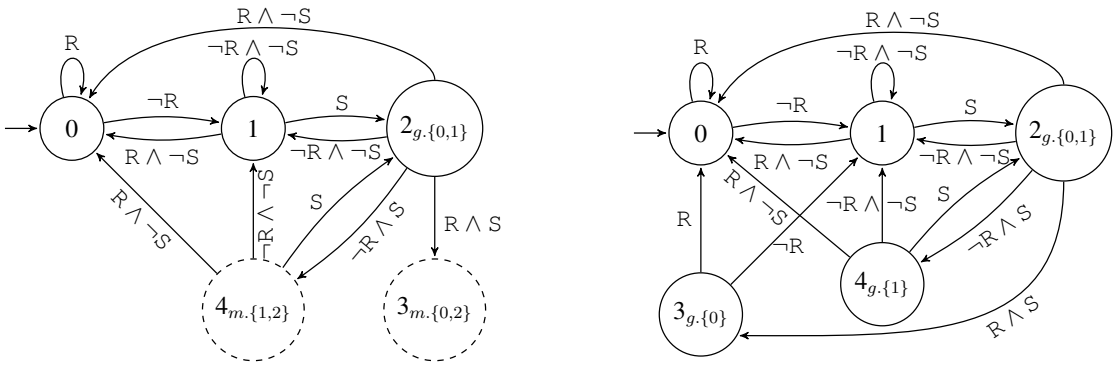


Figure 5.7: Constructing the violation monitor $\mathcal{U}(R, S)$: Replacing edge $1 \to 2$ with $1 \to \{0, 1\}$.

states is $\{0, 2\}$. Since 2 is an unsafe state, we go back to Line 3 and replace it again. This turns 3_m into 3_g and the resulting automaton is shown in Fig. 5.7 (b). At this moment, all error states (state 2) are eliminated and therefore $\mathcal{U}$ is fully constructed.

Unsafe States of $\mathcal{E} = \mathcal{U} \circ \mathcal{T}$

The error-avoiding monitor $\mathcal{E}$ is a synchronous composition of $\mathcal{U}$ and $\mathcal{T}$, where the unsafe states are defined as the union of the following sets:

- $\{(s, B) \mid s$ is a safe state in $\mathcal{U}$ coming from $\varphi^s\}$,
- $\{(s_m, B) \mid s_m$ results from merging edges and it contains no unsafe state$\}$, and
- $\{(s_g, A) \mid s_g$ results from replacing some unsafe states$\}$.

The reason is, when s is a safe state and s_m contains only safe states, the specification φ^s is not violated and therefore we must ensure $O' = O$ (state A in $\mathcal{T}$). In contrast, since s_g is created by replacing some originally unsafe states, the specification $\varphi^s(I, O)$ is violated, in which case $O' \neq O$ in order to avoid the violation of $\varphi^s(I, O')$. Figs. 5.8-5.10 show the resulting error-avoiding automaton. For brevity, only safe states and edges among these states are shown in Fig. 5.10. Note

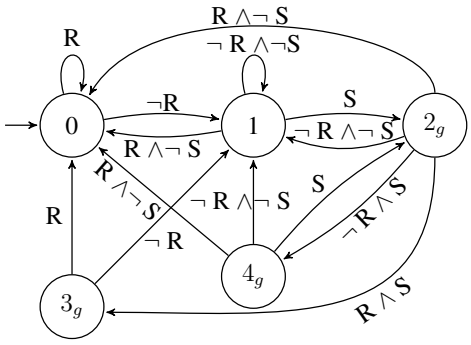


Figure 5.8: Violation monitor $\mathcal{U}(R, S)$.

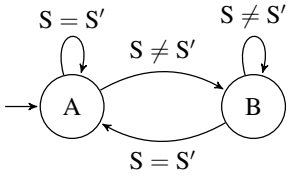


Figure 5.9: Deviation monitor $\mathcal{T}(S, S')$.

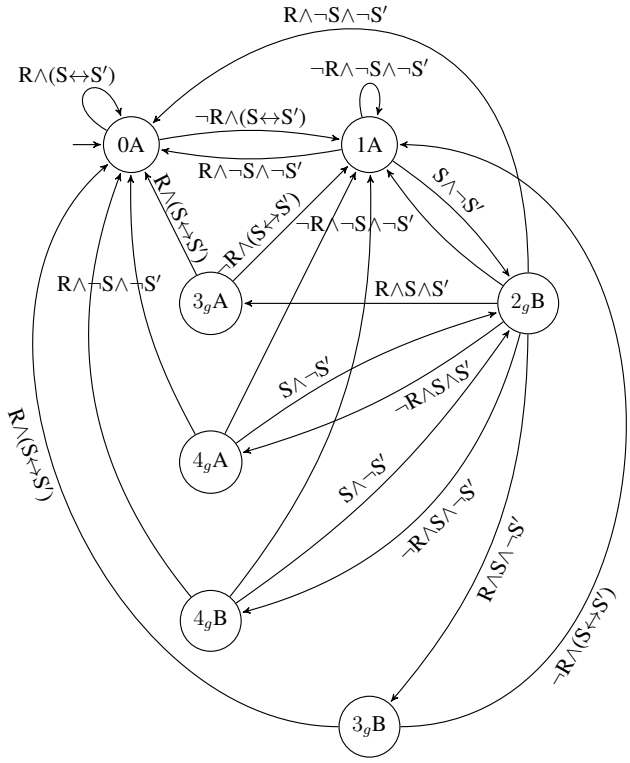


Figure 5.10: Error-avoiding monitor $\mathcal{E}(R, S, S')$.

that 2_gB, 3_gB, 4_gB are there because they are created by replacing some unsafe states and $O' \neq O$ holds in the B states.

Fig. 5.11 shows the game graph $\mathcal{G} = \mathcal{Q} \circ \mathcal{E}$ for the correctness monitor $\mathcal{Q}$ in Fig. 5.6 (b) and the error-avoiding monitor $\mathcal{E}$ in Fig. 5.10. For brevity, only the safe states in $\mathcal{G}$ and edges among these states are shown in Fig. 5.11. A safe state in $\mathcal{G}$ is a state $(g_{\mathcal{Q}}, g_{\mathcal{E}})$ where $g_{\mathcal{Q}}$ is safe in $\mathcal{Q}$ and $g_{\mathcal{E}}$ is safe in $\mathcal{E}$. The winning strategy of this safety game is denoted as $\rho = (\delta_\rho, \lambda_\rho)$, where δ_ρ is the transition function capturing a subset of the edges in Fig. 5.11, and λ_ρ is the output function determining the value of S' based on the current state and values of R and S. The shield $\mathcal{S}(R, S, S')$ is a reactive system that implements function δ_ρ and λ_ρ of ρ.

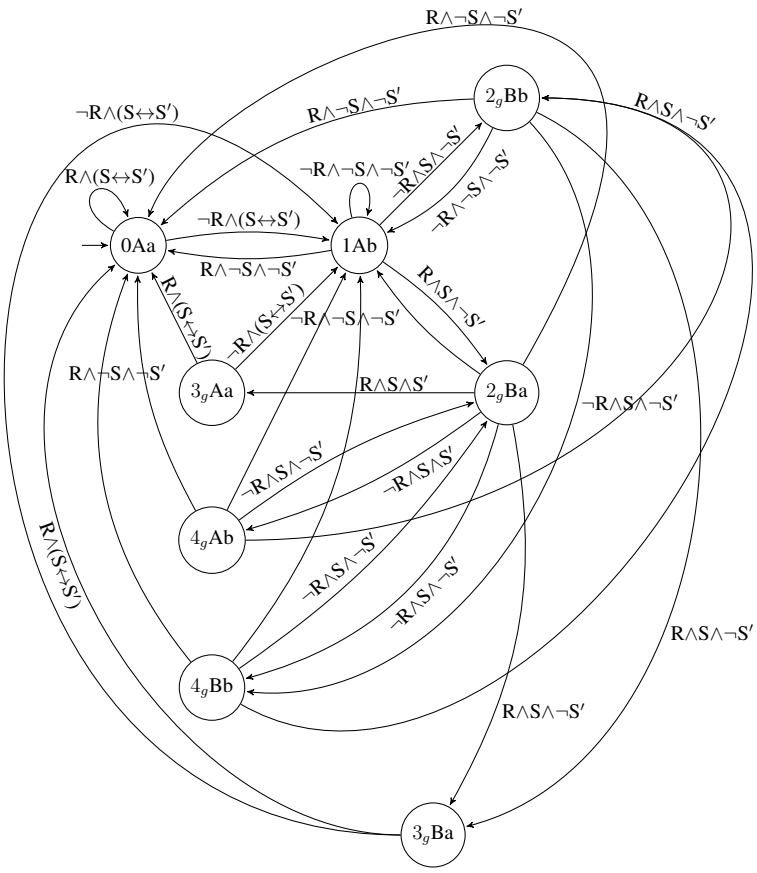


Figure 5.11: Game graph $\mathcal{G}(R, S, S')$, which is the composition of $\mathcal{Q}(R, S')$ and $\mathcal{E}(R, S, S')$.

5.3 Solving the Safety Game

We compute the winning strategy $\rho = (\delta_\rho, \lambda_\rho)$ by solving the two-player safety game $\mathcal{G} = (G, g_0, \Sigma, \Sigma_{O'}, \delta, F)$, where G is a finite set of game states, $g_0 \in G$ is the initial state, $F \subseteq G$ are the final (unsafe) states, $\delta : G \times \Sigma \times \Sigma_{O'} \to G$ is a complete transition function. The two players of the game are the shield and the environment (including the design $\mathcal{D}$). In every game state $g \in G$, the environment first chooses an input letter $\sigma \in \Sigma$, and then the shield chooses some output letter ${\sigma_O}' \in \Sigma_{O'}$, leading to the next state $g' = \delta(g, \sigma, {\sigma_O}')$. The sequence $\overline{g} = g_0 g_1 \ldots$ of

game states is called a *play*. We say that a play is *won* by the shield if and only if, for all $i = 0, 1, \ldots$ we have $g_i \in G \setminus F$.

5.3.1 Fix-point Computation

In this work, we use the algorithm of Mazala [109] to solve the safety game. In this algorithm, we compute "attractors" for a subset of safe states $(G \setminus F)$ and final states (F), until reaching the fix-point. Specifically, we maintain two sets of states: $\mathcal{F}$ is the set of states from which the shield will inevitably lose, and $\mathcal{W}$ is the set of states from which the shield has a strategy to win. We also define a function

$$MX(Z) = \{q \mid \exists \sigma \in \Sigma \,.\, \forall {\sigma_O}' \in \Sigma_{O'} \,.\, q' = \delta(q, \sigma, {\sigma_O}') \wedge (q' \in Z)\}$$

That is, $MX(Z)$ is the set of states from which the environment can force the transition to a state in Z regardless of how the shield responds.

The fix-point computation starts with $\mathcal{W} = G \setminus F$ and $\mathcal{F} = F$. In each iteration, $\mathcal{W} = \mathcal{W} \setminus MX(\mathcal{F})$ and $\mathcal{F} = \mathcal{F} \cup MX(\mathcal{F})$.

The computation stops when both $\mathcal{W}$ and $\mathcal{F}$ reach the fix-point.

5.3.2 Optimization

The computation of the winning strategy ρ in the safety game $\mathcal{G} = \mathcal{E} \circ \mathcal{Q}$ is time-consuming. In this section, we propose a new method for speeding up this computation. First, we note that a safe state in $\mathcal{G}$ must be safe in both $\mathcal{E}$ and $\mathcal{Q}$, meaning that a winning play in $\mathcal{G}$ must be winning in both of the subgames $\mathcal{E}$ and $\mathcal{Q}$. Therefore, instead of directly computing the winning region $\mathcal{W}$ of $\mathcal{G}$, which can be expensive due to the size of $\mathcal{G}$, we first compute the winning region $\mathcal{W}_1$ of the smaller subgame

$\mathcal{G}_1 = \mathcal{E}$, then compute the winning region $\mathcal{W}_2$ of the smaller subgame $\mathcal{G}_2 = \mathcal{Q}$, and finally compute the winning region $\mathcal{W}$ of the game $\mathcal{G}$ by using $\mathcal{W}_1 \times \mathcal{W}_2$ as the starting point. Since a winning play in $\mathcal{G}$ is winning in both $\mathcal{G}_1$ and $\mathcal{G}_2$, we know $\mathcal{W} \subseteq \mathcal{W}_1 \times \mathcal{W}_2$.

Furthermore, due to the unique characteristics of the subgames $\mathcal{G}_1 = \mathcal{E}$ and $\mathcal{G}_2 = \mathcal{Q}$, in practice, $\mathcal{W}_1 \times \mathcal{W}_2$ is often close to the final fix-point $\mathcal{W}$. This is because both $\mathcal{E}(I, O, O')$ and $\mathcal{Q}(I, O')$ are derived from the specification automaton φ^s. Specifically, each state in $\mathcal{Q}$ is simply a copy of the corresponding state in φ^s, whereas each state in $\mathcal{E}$ is either a copy of a safe state s in φ^s, or a new abstract state s_g that replaces some unsafe states in φ^s, or a new abstract state s_m consisting of only safe states in φ^s. Since it is cheaper to compute $\mathcal{W}_1$ and $\mathcal{W}_2$, this optimization can significantly speed up the fix-point computation.

5.4 Related Work

As we have already mentioned, our method for ensuring that the design $\mathcal{D}$ always satisfies the safety specification φ^s differs from both model checking [41, 130], which checks whether $\mathcal{D} \models \varphi^s$ but does not enforce φ^s, and reactive synthesis [30, 53, 129], which synthesizes the design $\mathcal{D}$ from a complete specification. Since our method is agnostic to the size and complexity of $\mathcal{D}$, it can be significantly more scalable than reactive synthesis in practice. Our method differs from the existing shield synthesis method of Bloem et al. [31] in that it can robustly handle burst error.

Our shield is a reactive system that can respond to a safety violation instantaneously, e.g., in the same clock cycle where the violation occurs, and therefore differs from the many existing methods for enforcing temporal properties [63, 97, 139] that have to buffer the erroneous output before correcting them. Similarly, it differs from the method by Luo and Rosu [103] for enforcing temporal logic properties in concurrent software, which relies on delaying the execution of one or more threads to avoid unsafe states. It also differs from the method by Yu et al. [168], which aims at

minimizing the edit-distance between two strings, but requires the entire input string to be available prior to generating the output string.

Renard et al. [136] proposed a runtime enforcement method for timed-automaton properties, but the method differs from ours as it assumes that the controllable input events can be delayed or suppressed, whereas our method relaxes such an assumption. Bauer et al. [26] and Falcone et al. [63] also studied what type of temporal logic properties can or cannot be monitored and enforced at run time. These works are orthogonal and complementary to ours. In this work, we focus on enforcing safety specification only. We leave the enforcement of liveness properties for future work.

Chapter 6

Runtime Enforcement for Real-Valued Signals

The shield synthesis technique presented in the previous chapter works only in the Boolean domain, by assuming that all input and output signals of the system, as well as the variables used in φ, are Boolean.

However, signals in cyber-physical systems may have real values and need to satisfy constraints such as $x + y \leq 1.53$. Naively treating the real-valued constraint as a predicate, or a Boolean variable P, may lead to *loss of information* at the synthesis time and *unrealizability* at run time. For example, while the Boolean combination $P \wedge \neg Q \wedge \neg R$ may be allowed, the corresponding real-valued constraint may not have solution, e.g., with $P : x + y \leq 1.53$, $Q : x < 1.0$ and $R : y < 1.0$. Therefore, a straightforward combination of the Boolean-level shield synthesis techniques with generic constraint solving at run time does not always work.

Even the use of *abstraction refinement* to combine a Boolean shield with constraint solving does not work. For example, one may be tempted to block $P \wedge \neg Q \wedge \neg R$ and ask the shield to generate

a new solution. However, since the shield must be reflexive, i.e., producing O' in the same clock cycle when the erroneous O occurs, it may be too slow to recompute a solution. Even if it is fast enough, the new solution may still be unrealizable in the real domain. In general, it is difficult to bound *a priori* the number of iterations in such an *abstraction-refinement* loop to meet the strict timing requirement.

In this chapter, we propose a shield synthesis method to guarantee, with certainty, the realizability of real-valued signals. Generally speacking, this is accomplished by treating Boolean and real-valued signals uniformly by adding a set of new constraints. These constraints take the form of two automata: a *relaxation automaton*, to capture the impossible combinations of predicates over signals in I and O, and a *feasibility automaton*, to capture the infeasible combinations of signals in O'. We use them to restrict the synthesis algorithm formulated as a two-player safety game, where the *antagonist* controls the erroneous O and the *protagonist* (shield) controls the corrected O': the game is won if the protagonist ensures that $\varphi(I, O')$ holds even if $\varphi(I, O)$ fails.

As shown by the aforementioned overall flow Fig. 1.2, where the input consists of real-valued I_r and O_r signals and a safety property φ_r defined over these signals. Internally, the shield $\mathcal{S}$ has three subcomponents: a converter from real-valued I_r/O_r signals to Boolean I/O signals, a converter from Boolean O' signals to real-valued O'_r signals, and a Boolean shield $\mathcal{S}(I, O, O')$. Note that the system, denoted $\mathcal{D}(I_r, O_r)$, is not required to synthesize the shield: by treating $\mathcal{D}$ as a blackbox, we ensure that $\mathcal{D} \circ \mathcal{S} \models \varphi_r$ for any $\mathcal{D}$.

Our shield synthesis algorithm first computes a set $\mathcal{P}$ of predicates over real-valued signals from φ_r, I_r, O_r and O'_r. Next, it leverages $\mathcal{P}$ to construct the Boolean abstractions φ, I, O and O', as well as the relaxation automaton $\mathcal{R}(I, O)$ and the feasibility automaton $\mathcal{F}(O')$. Using these components, it constructs and solve a safety game where the antagonist is free to introduce errors to O and the protagonist must correct them in O'. The winning strategy computed for the protagonist is the Boolean shield $\mathcal{S}(I, O, O')$. At run time, real values are computed for signals in O'_r by solving a

conjunction of constraints based on the Boolean values of signals in O.

To speed up the computation of real values at run time, we also propose a set of design-time optimizations, which leverage the information gathered from the shield to simplify the constraints to be solved at run time. When there are multiple real-valued solutions, the utility function γ shown in Fig. 1.2, which defines a *robustness* criterion, is used to pick the best one. We also propose a two-phase, *predict-and-validate* technique to speed up the computation of the real-valued solutions.

6.1 Technical Challenges

Using a Boolean shield to generate real-valued correction signals have two problems: realizability of the Boolean predicates, and quality of the real-valued signals.

6.1.1 Realizability of the Boolean Predicates

Conside the following real-valued LTL properties, which are abstractions of properties of an automotive powertrain control system [83] expressed in Signal Temporal Logic (STL [105]).

$$\mathsf{G}\big(l = \mathsf{power} \Rightarrow |\mu| < 0.2\big)$$
$$\mathsf{G}\Big(l = \mathsf{power} \wedge \mathsf{X}(l = \mathsf{normal}) \Rightarrow \mathsf{G}\big(|\mu| < 0.02\big)\Big)$$

The input signal l denotes the system mode, which may be normal or power. The output signal μ is the normalized error of the air-fuel (A/F) ratio inside an internal combustion engine. Let λ be the A/F ratio and λ_{ref} be a reference value, then $\mu = (\lambda - \lambda_{ref})/\lambda_{ref}$. Since it affects gas emission, driveability and fuel efficiency, it must be kept in certain regions depending on the system mode.

The first property says that $|\mu|$ should stay below 0.2 in the power mode. The second property

says that, after the system changes from the power model to the normal mode, $|\mu|$ should stay below 0.02. In the Boolean versions, A denotes whether the system is in the power mode, while B_1 and B_2 denote $|\mu| < 0.2$ and $|\mu| < 0.02$, respectively. The combination $\neg B_1 \wedge B_2$ is unrealizable, because $|\mu|$ cannot be both greater than 0.2 and less than 0.02.

However, the shield synthesized by existing methods is not aware of this problem, and thus may produce combinations of Boolean values that are not realizable in the real domain. If the shield's input is $\neg A \wedge \neg B_1 \wedge \neg B_2$, the shield's output will be $\neg B_1' \wedge B_2'$, despite that $|\mu'| >= 0.2 \wedge |\mu'| < 0.02$ is unsatisfiable.

We solve this problem by checking the compatibility of the predicates at the synthesis time, to guarantee the realizability of these predicates at run time. Details will be presented in Section 6.2.

6.1.2 Quality of the Real-valued Output

Even if the Boolean values are realizable, the real-valued solution may not be of high quality when the solution is computed by a generic LP solver. Assume that all predicates are linear constraints, the output of a Boolean shield would be a conjunction of linear constraints. As illustrated in Fig. 1.2, the back-end may convert O', the Boolean shield's output, to O'_r, the real-valued output by solving a linear programming (LP) problem.

However, it may not produce a reasonable output. Consider $\mathsf{G}(A \Rightarrow B)$, which abstracts $\mathsf{G}\big(l = \mathsf{power} \Rightarrow |\mu| < 0.2\big)$. Suppose the original system's output violates the property $|\mu| < 0.2$ as shown by the blue line in Fig. 6.1, where the two erroneous values are in the middle. The correction computed by an LP solver may be any of the infinitely many values in the interval (-0.2, +0.2), including -0.19 and 0. However, neither of these two values may be acceptable in a real system, which expects the signal to be *stable*, not *arbitrary*.

Ideally, we want to generate real-valued signals that are smooth and consistent with physical laws

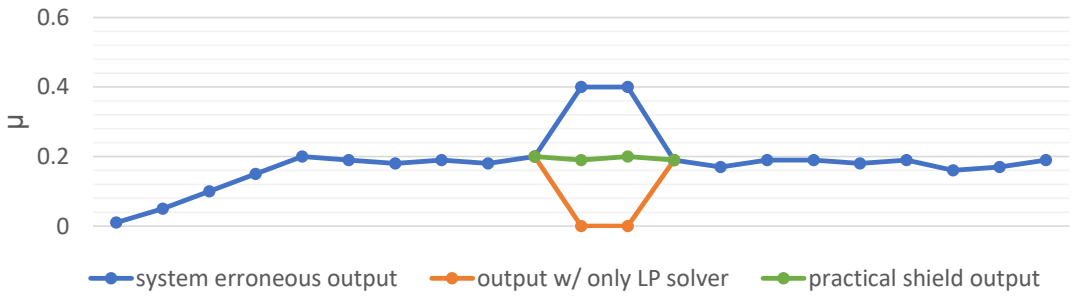


Figure 6.1: Importance of the smoothness in real-valued correction signals.

of the environment, e.g., the green line in Fig. 6.1. Toward this end, we leverage a utility function, γ, to impose *robustness* in addition to *correctness* constraints. With both types of constraints, the LP solver can generate values of high quality.

We also propose a technique to speed up the computation of these real values. The intuition is that system dynamics may be approximated using (linear or non-linear) regression, which predicts the current value of a signal based on its values in the recent past. Thus, we develop a fast prediction unit to guess the value, followed by a fast validation unit to check its validity. If the predicted value is valid, it will serve as the shield's output. Otherwise, we invoke the LP solver. Details will be presented in Section 6.3.

6.2 Synthesizing the Boolean Shield

In this section, we present our method for ensuring the realizability of the Boolean shield's correction signals. The idea is to check the compatibility of predicates inside the game-based algorithm for synthesizing the shield. To improve efficiency, we check predicate combinations only when they are involved in compute the winning strategy.

Algorithm 8 shows the procedure, where blue highlighted lines address the *realizability* issue, while the remainder follows the classic algorithm in the prior work [31, 88, 163]. First, it creates $\mathcal{P}$,

Algorithm 8 Synthesizing a realizable Boolean shield $\mathcal{S}_{bool}$ from φ_r.

Let $\mathcal{P}$ be the set of predicates over real-valued variables in φ_r;
Let φ, I, O, O' be Boolean abstractions of φ_r, I_r, O_r, O'_r via $\mathcal{P}$;
function SYNTHESIZEBOOL ($\mathcal{P}$, I, O, O')
    $\mathcal{Q}(I, O') \leftarrow$ GENCORRECTNESSMONITOR(φ)
    $\mathcal{E}(I, O, O') \leftarrow$ GENERRORAVOIDINGMONITOR(φ)
    $\mathcal{G} \leftarrow \mathcal{Q} \circ \mathcal{E}$
    $\mathcal{W} \leftarrow$ COMPUTEWINNINGSTRATEGY($\mathcal{G}$)
    $\mathcal{R}(I, O) \leftarrow$ **GENRELAXATIONAUTOMATON**($P, I, O, \mathcal{W}$)
    $\mathcal{F}(O') \leftarrow$ **GENFEASIBILITYAUTOMATON**($\mathcal{R}$)
    $\mathcal{G}_r \leftarrow \mathcal{W} \circ \mathcal{R} \circ \mathcal{F}$
    $\omega_r \leftarrow$ **COMPUTEWINNINGSTRATEGY**($\mathcal{G}_r$)
    $\mathcal{S}_{bool}(I, O, O') \leftarrow$ IMPLEMENTSHIELD(ω_r)
    return $\mathcal{S}_{bool}$
end function

the set of predicates from the real-valued specification φ_r. Then, it uses $\mathcal{P}$ to compute a Boolean abstraction of φ_r, denoted φ. Next, it uses φ to formulate a two-player safety game $\mathcal{G}$ where the antagonist controls I and O, the protagonist controls O', and $\mathcal{W}$ is the winning region where the protagonist may win the game.

Since the construction of the safety game $\mathcal{G}$ is part of the prior work and is well understood, we refer to Bloem et al. [31] and Meng et al. [163] for details. Here, it suffices to say that $\mathcal{G}$ is a synchronous composition of $\mathcal{E}$, an error-avoiding monitor that outlines all possible ways in which the antagonist may introduce errors in O and the protagonist may introduce corrections in O', and $\mathcal{Q}$, a correctness monitor that ensures $\varphi(I, O')$ always holds.

Since a winning strategy in $\mathcal{W}$ may not be realizable in the real domain, our next step is to compute a strategy ω_r based on $\mathcal{W}$ while ensuring correction signals produced by ω_r are always realizable. Toward this end, we introduce two additional automata: the *feasibility* automaton $\mathcal{F}(O')$ and the *relaxation* automaton $\mathcal{R}(I, O)$. Specifically, $\mathcal{F}$ is used to identify and remove the infeasible edges in ω, i.e., corrections in O' with no real-valued solutions. $\mathcal{R}$ is used to identify and remove the unrealistic errors in I and O, i.e., errors that are impossible and will not occur in the first place.

In other words, $\mathcal{F}$ restricts the search to realizable solutions, and $\mathcal{R}$ allows us to have less worry and more freedom while computing the winning strategy. Thus, the new game $\mathcal{G}_r$ is a composition of $\mathcal{W}$, $\mathcal{R}$ and $\mathcal{F}$. Based on the winning strategy ω_r computed from $\mathcal{G}_r$, we can construct a shield $\mathcal{S}_{bool}$ that is guaranteed to be realizable at run time.

In the remainder of this section, we illustrate the details while focusing on the highlighted lines in Algorithm 8.

6.2.1 Computing the Predicates

$\mathcal{P}$ is the set of predicates over real-valued signals used in φ_r, where φ_r is expressed in Signal Temporal Logic (STL). In addition to the LTL operators, STL also has dense time intervals associated with temporal operators and constraints over real-valued variables.

Consider the STL formulas below, which come from the powertrain control system [83] without modification.

$$\mathsf{G}_{[\tau_s,T]}\big(l=\mathsf{power} \Rightarrow |\mu| < 0.2\big)$$
$$\mathsf{G}_{[\tau_s,T]}\Big(l=\mathsf{power} \wedge \mathsf{X}(l=\mathsf{normal}) \Rightarrow \mathsf{G}_{[\eta,\frac{\varsigma}{2}]}\big(|\mu| < 0.02\big)\Big)$$

Here, $\mathsf{G}_{[\tau_1,\tau_2]}$ is the temporal operator augmented with time interval $[\tau_1, \tau_2]$, l is the system mode, and μ is the normalized error of the air-fuel ratio. The first property says that $|\mu|$ should stay below 0.2 immediately after the system switch to the power mode, i.e., between time τ_s and time T. The second property says that, when it switches from the power mode to the normal mode, $|\mu|$ should settle down to below 0.02 after time η and before time $\frac{\varsigma}{2}$.

To compute $\mathcal{P}$, first, we convert each time interval to a conjunction of linear constraints, e.g., by using a time variable t to represent the bounds in intervals $[\tau_s, T]$ and $[\eta, \frac{\varsigma}{2}]$.

T_1: $(t \geq \tau_s)$ T_2: $(t \leq T)$

T_3: $(t \geq \eta)$ T_4: $(t \leq \frac{\varsigma}{2})$

Next, we convert the constraints over real-valued variables to predicates. From the running example, we will produce the following predicates:

L_1: $(l = \mathsf{power})$ L_2: $(l = \mathsf{normal})$

M_1: $(|\mu| < 0.2)$ M_2: $(|\mu| < 0.02)$

6.2.2 Computing the Boolean Abstractions

After the set $\mathcal{P}$ of predicates is computed, we use it to compute the Boolean abstractions of φ_r, I_r, O_r and O'_r. This step is straightforward. To compute φ from φ_r, we traverse the abstract syntax tree (AST) of φ_r and, for each AST node n that corresponds to a real-valued predicate $P \in \mathcal{P}$, we replace P with a new Boolean variable v_P.

To compute I from I_r, we traverse the predicates in $\mathcal{P}$ and, for each predicate $Q \in \mathcal{P}$ defined over some real-valued signals in I_r, we add a new Boolean variable v_Q to I. Similarly, O and O' are also computed from O_r and O'_r by creating new Boolean variables.

6.2.3 Computing the Relaxation Automaton

The relaxation automaton $\mathcal{R}$ aims to identify impossible combinations of I and O values, and since they will never occur in the shield's input, there is no need to make corrections in the shield's output. There may be two reasons why a value combination is impossible:

1. The values of real-valued predicates are incompatible, e.g., as in $|\mu| < 0.02$ and $|\mu| > 0.2$.

2. The values are not consistent with physical laws of the environment, e.g., time never travels

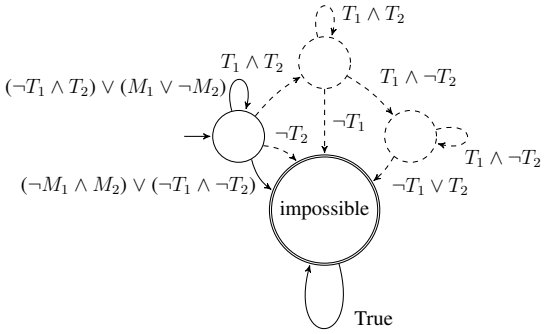


Figure 6.2: Relaxation automaton $\mathcal{R}(I, O)$: *impossible* means the system $\mathcal{D}$ will not allow the state to be reached, and the shield $\mathcal{S}$ can treat it as *don't care*.

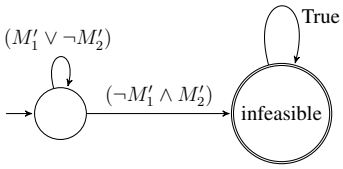


Figure 6.3: Feasibility automaton $\mathcal{F}(O')$: *infeasible* means the state is unrealizable, and the shield $\mathcal{S}$ must avoid the related edges while generating solutions.

backward. For example, with respect to the time interval $[\tau_s, T]$, the transition from $T_1 \wedge T_2$ to $\neg T_1 \wedge T_2$ is impossible.

In addition, our method allows users to provide more constraints to characterize physical laws of the environment or their understanding of the behaviors of the system $\mathcal{D}$.

States in the relaxation automaton $\mathcal{R}$ are divided into two types: *normal* states and *impossible* states. Here, *normal* means the I/O behavior of the system $\mathcal{D}$ may occur, whereas *impossible* means it will never occur. Since impossible I/O behavior will never occur in the shield's input, the shield may treat it as *don't-care* and thus have more freedom to compute the winning strategy.

Example Fig. 6.2 shows the relaxation automaton for our running example. Here, the dashed edges come from the physical laws (time never travels backward), while the solid edges comes from the compatibility of real-valued predicates defined over l and μ. In particular, the combination $\neg M_1 \wedge M2$ is identified as impossible, because $|\mu|$ cannot be greater than 0.2 and less than 0.02 at the same time.

To check the compatibility of the predicate values, conceptually, one can iterate through all possible value combinations for the predicates in $\mathcal{P}$, and check each combination with an LP solver. If the

combination is *unsatisfiable (UNSAT)* according to the LP solver, we say it is impossible. However, in our actual implementation, the compatibility checking is performed significantly more efficiently, due to the use of variable partitioning and UNSAT cores. First, $\mathcal{P}$ may be divided into subgroups, such that predicates from different subgroups do not interfere with each other. Therefore, value combinations may be computed via Cartesian products. Second, when a value combination is proved to be unsatisfiable, we compute its UNSAT core, i.e., a minimal subset that itself is UNSAT. By leveraging these UNSAT cores, we can significantly speed up the checking of value combinations.

6.2.4 Computing the Feasibility Automaton

The feasibility automaton $\mathcal{F}$ aims to capture the combinations of O' values that are unrealizable in the real domain. Similar to $\mathcal{R}$, states in $\mathcal{F}$ are divided into two types: *safe* and *unsafe*. Here, *safe* means the value combinations are realizable in the real domain, whereas *unsafe* means the value combinations are unrealizable.

Fig. 6.3 shows an example of the feasibility automaton for our running example. In this case, all predicates are the primed versions, because they are defined over the O' signals, which are part of the modified output of the shield. Upon $\neg M_1' \wedge M_2'$, the automaton goes into the unsafe state, because this particular value combination is unsatisfiable.

During the computation of the winning strategy ω_r, we need to make sure that such unsafe states are avoided.

6.2.5 Solving the New Safety Game

The new safety game $\mathcal{G}_r$ is defined as the composition of $\mathcal{W}$, the winning region of the Boolean game $\mathcal{G}$, the relaxation automaton $\mathcal{R}$, and the feasibility automaton $\mathcal{F}$. We tweak the winning region

automaton $\mathcal{W}$ by adding an *unsafe* state for all edges going out of $\mathcal{W}$. Here, composition means the standard synchronous product, where a state transition exists only if it is allowed by all three components ($\mathcal{W}$, $\mathcal{R}$ and $\mathcal{F}$). Furthermore, safe states of $\mathcal{G}_r$ are defined as either (1) states that are both *safe* in $\mathcal{W}$ and *feasible* in $\mathcal{F}$, or (2) states that are *impossible* in $\mathcal{R}$.

More formally, assume that F^w is the set of unsafe states of the winning region $\mathcal{W}$, F^f is the set of infeasible states of the feasibility automaton $\mathcal{F}$, and F^r is the set of the impossible states of the relaxation automaton $\mathcal{R}$. The set of safe states in the new game $\mathcal{G}_r$ is defined as $(\neg F^w \wedge \neg F^f) \vee F^r$.

Finally, we solve $\mathcal{G}_r$ using standard algorithms for safety games, e.g., Mazala [109], which are also used in the prior work [31, 88, 163]. The result is a winning strategy ω_r, which in turn may be implemented as a reactive component $\mathcal{S}_{bool}$. Note that $\mathcal{S}_{bool}$ is a Mealy machine that takes I and O signals as input and returns the modified O' signals as output. Furthermore, due to the use of $\mathcal{R}$ and $\mathcal{F}$, the output of $\mathcal{S}_{bool}$ is guaranteed to be realizable at run time.

6.3 Generating the Real-valued Signals

In this section, we present our method for computing the real-valued signals (O'_r) at run time, based on the Boolean shield's output (O').

Algorithm 9 shows the details of our method, which needs I_r, O_r, O'_r, the set $\mathcal{P}$ of predicates, $\mathcal{S}_{bool}$, and a utility function γ, which is used to evaluate the quality of the real-valued solution. First, real values in I_r and O_r are transformed to Boolean values in I and O. Then, they are used by $\mathcal{S}_{bool}$ to compute new values in O'. When O' and O have the same Boolean value, meaning the shield does not make any correction, O'_r and O' will also have the same real value; in this case, no computation is needed (Line 5). However, when O' and O have different values, we need to recompute the real values in O'_r (Lines 7-11).

Algorithm 9 Computing real-valued correction signals at run time.

function COMPUTEREALVALUES(I_r, O_r, O'_r, $\mathcal{P}$, $\mathcal{S}_{bool}$, γ)
  $I, O \leftarrow$ GENBOOLEANABSTRACTION($I_r, O_r, \mathcal{P}$)
  $O' \leftarrow$ GENBOOLEANSHIELDOUTPUT($\mathcal{S}_{bool}, I, O$)
  if $O' = O$ **then**
    $O'_r = O_r$
  else
    $O'_r \leftarrow$ **PREDICTION**($Hist$)
    if $\neg$ **SATISFIABLE**($\mathcal{P}, O', O'_r$) **then**
      $model \leftarrow$**LPSOLVE**($\mathcal{P}, \gamma, O'$)
      $O'_r \leftarrow model$
    end if
  end if
  $Hist \leftarrow Hist \cup \{O'_r\}$
end function

6.3.1 Robustness Optimization

Since the output of the Boolean shield is an assignment of the Boolean predicates in O', and each predicate corresponds to a linear constraint of the form $\Sigma_{i=1}^{k} a_i x_i \leq 0$, conceptually, the real values in O'_r can be computed by solving the linear programming (LP) problem.

However, naively invoking the LP solver does not guarantee that the real-valued solution is of high quality. Instead, we develop the following optimization to improve the quality of the solution. Specifically, we restrict the LP problem using a robustness constraint derived from the utility function γ. While there may be various ways of defining robustness, especially in the context of STL [50, 62], a straightforward way that works in practice is to ensure the signal is *smooth* (see the example in Fig. 6.1).

That is, we restrict the LP problem using the objective function as follows:

$$\min \left(|val^i - \frac{\sum_{k=1}^{N} val^{i-k}}{N}| \right)$$

where val^i denotes the current value (at the i-th time step), val^{i-k}, where $k = 1, 2, \ldots$, denotes the

value in the recent past, and the above function aims to minimize its distance between val^i and the average of the previous N values, stored in $Hist$ (Line 13).

6.3.2 Value Prediction and Validation

While the robustness constraint improves the quality of the real-valued solution, it also increases the computational cost of LP solving. To reduce the computational cost, we develop a two-phase technique for computing the solution.

First, we predict the value of a signal using standard regression algorithms based on the historical values of the signal in the immediate past (Line 7 in Algorithm 9). Here, the procedure PREDICTION leverages historical values stored in $Hist$. Since the signal is expected to be *smooth*, standard linear or non-linear regression can be very accurate in practice.

Next, we validate the predicted value (Line 8). This is accomplished by plugging the predicted value for O'_r into the combination of Boolean predicates defined by $\mathcal{P}$ and the values of signals in O'. If it is valid, the value is accepted as the final output, and invocation of the LP solver is avoided. Note that the time taken to perform prediction and validation is significantly smaller than that of the LP solving.

Only when the predicted value is not valid, we invoke the LP solver (Line 9). Even in this case, the response time is fast because we use the same LP solver for validation and LP solving. Due to the incremental computation inside the solver, the solution used for validation, which is often close to the final solution, can help speed up the LP solving.

6.4 Related Work

We are the first to synthesize real-valued shields and demonstrate their application to cyber-physical systems. As we have mentioned earlier, prior work on shield synthesis has been restricted to the Boolean domain. Specifically, Bloem et al. [31] introduced the notion of shield together with a synthesis algorithm, which minimizes the deviation between O and O' under the assumption that *no two errors occur within k steps*. Wu et al. [163] improved the algorithm to deal with *burst error*. That is, if more errors occur within the k-step recovery period, instead of entering a *fail-safe* state, they keep minimizing the deviation. Könighofer et al. [88] further improved the shield while Alshiekh et al. [17] leveraged it to improve the performance of reinforcement learning. However, none of the existing techniques dealt with the realizability problems associated with real-valued signals.

There is also a large body of work on reactive synthesis [30, 53, 129, 146] and controller synthesis [64, 101, 131, 132]. The goal is to synthesize $\mathcal{D}$ from a complete specification Ψ, or the control sequences for $\mathcal{D}$ to satisfy Ψ. In both cases, the complexity depends on $\mathcal{D}$. This is more challenging, for two reasons. First, specifying all aspects of the system requirement may be difficult. Second, even if Ψ is available, synthesizing $\mathcal{D}$ from Ψ is difficult due to the inherent *double exponential* complexity of the synthesis problem. Our method, in contrast, treats $\mathcal{D}$ as a blackbox while focusing on a small subset $\varphi \subseteq \Psi$ of *safety-critical* properties. This is why shield synthesis may succeed where conventional reactive synthesis fails.

Renard et al. [134] proposed a runtime enforcement method for timed automata, but assumed that controllable input events may be delayed or suppressed, whereas our method does not require such an assumption. Bauer et al. [26] and Falcone et al. [63] studied various types of temporal logic properties that may be monitored or enforced at run time. Renard et al. [135] also leveraged Büchi games to enforce regular properties with uncontrollable events. Our work is orthogonal in that it

tackles the realizability and efficiency problems associated with real-valued signals. Furthermore, we focus on safety while leaving liveness and hyper-properties [32] for future work.

An important feature of the shield synthesized by our method is that it always makes corrections *instantaneously*, without any delay. Therefore, it differs from a variety of solutions that allow delayed corrections. In some cases, for example, buffers may be allowed to store the erroneous output temporarily, before computing the corrections [63, 97, 139]. In this context, the notion of *edit-distance* is more relevant. Yu et al. [168], for example, proposed a technique for minimizing the edit-distance between two strings, but the technique requires the entire input be stored in a buffer prior to generating the output. However, when the buffer size reduces to zero, these existing techniques would no longer work.

Runtime enforcement is related to, but different from, software techniques for error avoidance. For example, failure-oblivious computing [102, 137] was used to allow applications to execute through memory errors; temporal properties [103, 171] were leveraged to control thread schedules to avoid runtime failures of concurrent software. However, these techniques are not designed to target cyber-physical systems with real-valued signals, where corrections are expected to be made instantaneously, in the same clock cycle when the error occurs.

Chapter 7

Evaluation

The side channel detection and mitigation techniques presented in Chapters 2 and 3 have been implemented in a software tool named `SC-Eliminator`, based on the LLVM compiler platform. The tool has been evaluated on a number of cryptographic software programs. Similarly, the shield synthesis techniques presented in Chapters 5 and 6 have been implemented in software tools and evaluated on realistic systems such as automotive powertrain control and autonomous driving.

7.1 Timing Side Channel Elimination

Our detection of potential timing leaks is accomplished by three LLVM-based analysis passes: a sensitivity analysis that reads in the list of secret variables and propagates the sensitivity attribute to other variables via control- and data-dependencies; a static cache analysis invoked on demand to decide whether a store or load instruction definitely results in a cache hit; and a leakage detection pass leveraging results of the above two analyses to identify instructions that may cause timing leaks.

Our mitigation is accomplished by two LLVM-based transformation passes. The first pass aims to replace sensitive conditional statements with functionally-equivalent but time-invariant assignments. The second pass aims to mitigate accesses to sensitive look-up tables that, depending on the value of the index, may lead to different cache behaviors.

Our experiments aimed to answer three research questions: (1) Is our new method effective in detecting and mitigating instruction- and cache-timing leaks? (2) Is our method efficient in handling real-world cryptographic software? (3) Is the overhead of mitigated code low enough (in terms of code size and run time) for practical use?

7.1.1 Benchmarks

We conducted experiments on C/C++ programs that implement well-known cryptographic algorithms by compiling them to bit-code using Clang/LLVM. Table 7.1 shows the benchmark statistics. In total, there are 19,708 lines of code from libraries including a real-time Linux kernel (Chronos [48]), a lightweight cryptographic library (FELICS [2]), a system for performance evaluation of cryptographic primitives (SuperCop [5]), the Botan cryptographic library [1], three textbook implementations of cryptographic algorithms [142], and the GNU Libgcrypt library [6]. Columns 1 and 2 show the benchmark name and source. Column 3 shows the number of lines of code (LoC). Columns 4 and 5 show the number of conditional jumps (# IF) and the number of lookup tables (# LUT). The last two columns show more details of these lookup tables, including the total, minimum, and maximum table size.

7.1.2 Experimental Results: Leak Detection

Table 7.2 shows the results of applying our leak detection technique based on static analysis, where Columns 1-4 show the name of the benchmark together with the number of conditional jumps (# IF),

Table 7.1: Benchmark statistics.

Name	Description	# LoC	# IF	# LUT	LUT size in Bytes	
					total	(min, max)
aes	AES in Chronos [48]	1,379	3	5	16,424	(40, 4096)
des	DES in Chronos	874	2	11	6,656	(256, 4096)
des3	DES-EDE3 in Chronos	904	2	11	6,656	(256, 4096)
anubis	Anubis in Chronos	723	1	7	6,220	(76, 1024)
cast5	Cast5 cipher (rfc2144) in Chronos	799	0	8	8,192	(1024, 1024)
cast6	Cast6 cipher (rfc2612) in Chronos	518	0	6	4,896	(32, 1024)
fcrypt	FCrypt encryption in Chronos	401	0	4	4,096	(1024, 1024)
khazad	Khazad algorithm in Chronos	841	0	9	16,456	(72, 2048)
LBlock	LBlock cipher from Felics [2]	1,005	0	10	160	(16,16)
Piccolo	Piccolo cipher from Felics	243	2	4	148	(16,100)
PRESENT	PRESENT cipher from Felics	183	0	33	2,064	(15,64)
TWINE	TWINE cipher from Felics	249	0	3	67	(16,35)
aes	AES in SuperCop [5]	1099	4	10	8,488	(40, 1024)
cast	CAST in SuperCop	942	5	8	16,384	(2048, 2048)
aes_key	AES key_schedule in Botan [1]	502	3	4	8,704	(256,4096)
cast128	cast 128-bit in Botan	617	2	8	8,192	(1024,1024)
des	des cipher in Botan	835	1	12	10,240	(1024,2048)
kasumi	kasumi cipher in Botan	275	2	2	1,152	(128,1024)
seed	seed cipher in Botan	352	0	5	4,160	(64,1024)
twofish	twofish cipher in Botan	770	18	9	5,150	(32,1024)
3way	3way cipher reference [142]	177	10	0	0	(0,0)
des	des cipher reference	463	16	14	2,302	(16,512)
loki91	loki cipher reference	231	10	1	32	(32,32)
camellia	camellia cipher in Libgcrypt [6]	1453	0	4	4,096	(1024,1024)
des	des cipher in Libgcrypt	1486	2	13	2,724	(16,2048)
seed	seed cipher in Libgcrypt	488	3	5	4,160	(64,1024)
twofish	twofish cipher in Libgcrypt	1899	1	6	6,380	(256,4096)

lookup tables (# LUT), and accesses to table elements (# LUT-access), respectively. Columns 5-7 show the number of *sensitive* conditional jumps, lookup tables, and accesses, respectively. Thus, non-zero in the sensitive `#IF` column means there is instruction-timing leakage, and non-zero in the sensitive `#LUT-access` means there is cache-timing leakage. We omit the time taken by our static analysis since it is negligible: in all cases the analysis completed in a few seconds.

Although conditional statements (#IF) exist in many benchmarks, few are sensitive. Indeed, only `twofish` from Botan[1] and three old textbook implementations (`3way`, `des`, and `loki91`) have leaks of this type. In contrast, many lookup tables are sensitive. This result was obtained using a

Table 7.2: Results of conducting static leakage detection.

Name	Total			Sensitive (leaky)		
	# IF	# LUT	# LUT-access	# IF	# LUT	# LUT-access
aes	3	5	424	0	4	416
des	2	11	640	0	11	640
des3	2	11	1,152	0	11	1,152
anubis	1	7	871	0	6	868
cast5	0	8	448	0	8	448
cast6	0	6	448	0	4	384
fcrypt	0	4	128	0	4	128
khazad	0	9	240	0	8	248
*LBlock	0	10	320	0	0	0
*Piccolo	2	4	121	0	0	0
*PRESENT	0	33	1,056	0	0	0
*TWINE	0	3	156	0	0	0
aes	4	10	706	0	9	696
cast	5	8	448	0	8	448
aes_key	3	4	784	0	2	184
cast128	2	8	448	0	8	448
des	1	12	264	0	8	256
kasumi	2	2	192	0	2	192
seed	0	5	576	0	4	512
twofish	18	9	2,576	16	8	2,512
3way	10	0	0	3	0	0
des	16	14	456	2	8	128
loki91	10	1	512	4	0	0
camellia	0	4	32	0	4	32
des	2	13	231	0	8	128
seed	3	5	518	0	4	200
twofish	1	6	8,751	0	5	2,576

representative cache configuration: fully associative LRU cache with 512 cache lines, 64 bytes per line, and thus 32 Kilobytes in total.

Some benchmarks, e.g., `aes_key` from Botan [1], already preload lookup tables; however, our analysis still reports timing leakage, as shown in Figure 7.1, where `XEK` is key-related and used to access an array in the second for-loop. Although the table named `TD` is computed at run time (thus capable of avoiding `flush+reload` attack) and all other tables are preloaded before accesses, they forgot to preload `SE[256]`, which caused the cache-timing leak.

```
const uint8_t SE[256] = {0x63, 0x7C, 0x77, 0x7B,...};
void aes_key_schedule(const uint8_t key[], size_t length,
  std::vector<uint32_t>& EK, std::vector<uint32_t>& DK,
  std::vector<uint8_t>& ME, std::vector<uint8_t>& MD)
{
  static const uint32_t RC[10] = {0x01000000, 0x02000000,...};
  std::vector<uint32_t> XEK(48), XDK(48);
  const std::vector<uint32_t>& TD = AES_TD();

  for(size_t i = 0; i != 4; ++i)
    XEK[i] = load_be<uint32_t>(key, i);

  for(size_t i = 4; i < 44; i += 4) {
    XEK[i] = XEK[i-4] ^ RC[(i-4)/4] ^
             make_uint32(SE[get_byte(1, XEK[i-1])],
                  SE[get_byte(2, XEK[i-1])],
                  SE[get_byte(3, XEK[i-1])],
                  SE[get_byte(0, XEK[i-1])]);
    ...
  }
  ...
}
```

Figure 7.1: Reduction: preloading only in the first iteration.

7.1.3 Experimental Results: Leak Mitigation

To evaluate whether our method can robustly handle real applications, we collected results of applying our mitigation procedure to each benchmark. Table 7.3 shows the results. Specifically, Columns 2-5 show the result of our mitigation without cache analysis-based optimization, while Columns 6-9 show the result with the optimization. In each case, we report the number of LUT accesses actually mitigated, the time taken to complete the mitigation, the increase in program size, and the increase in runtime overhead. For `anubis`, in particular, our cache analysis showed that only 10 out of the 868 sensitive LUT accesses needed mitigation; as a result, optimization reduced both the program's size (from 9.08x to 1.10x) and its execution time (from 6.90x to 1.07x).

We also compared the execution time using generic (bitwise) versus optimized (CMOV) implementations of *CTSEL(c,t,e)*. Figure 7.2 shows the result in a scatter plot, where points below the

Table 7.3: Results of leakage mitigation. Runtime overhead is based on average of 1000 simulations with random keys.

Name	Mitigation w/o opt				Mitigation w/ opt			
	# LUT-a	Time(s)	Prog-size	Ex-time	# LUT-a	Time(s)	Prog-size	Ex-time
aes	416	0.61	5.40x	2.70x	20	0.28	1.22x	1.11x
des	640	1.17	19.50x	5.68x	22	0.13	1.23x	1.07x
des3	1,152	1.80	12.90x	12.40x	22	0.46	1.13x	1.07x
anubis	868	3.12	9.08x	6.90x	10	0.75	1.10x	1.07x
cast5	448	0.79	7.24x	3.84x	12	0.22	1.18x	1.07x
cast6	384	0.72	7.35x	3.48x	12	0.25	1.16x	1.08x
fcrypt	128	0.07	5.70x	1.59x	8	0.03	1.34x	1.05x
khazad	248	0.45	8.60x	4.94x	16	0.07	1.49x	1.35x
aes	696	0.96	9.52x	2.39x	18	0.22	1.21x	1.06x
cast	448	1.42	13.40x	6.50x	12	0.30	1.35x	1.20x
aes_key	184	0.27	1.35x	1.19x	1	0.23	1.00x	1.00x
cast128	448	0.42	3.62x	2.48x	12	0.10	1.09x	1.06x
des	256	0.21	3.69x	1.86x	16	0.06	1.17x	1.08x
kasumi	192	0.18	2.27x	1.37x	4	0.11	1.03x	1.01x
seed	512	0.57	6.18x	1.94x	12	0.15	1.12x	1.03x
twofish	2,512	29.70	5.69x	4.77x	8	10.6	1.02x	1.03x
3way	0	0.01	1.01x	1.03x	0	0.01	1.01x	1.03x
des	128	0.05	2.21x	1.22x	8	0.03	1.09x	1.11x
loki91	0	0.01	1.01x	2.83x	0	0.01	1.01x	2.83x
camellia	32	0.04	2.21x	1.35x	4	0.03	1.20x	1.09x
des	128	0.06	2.30x	1.20x	8	0.03	1.10x	1.02x
seed	200	0.01	1.38x	1.36x	8	0.01	1.20x	1.18x
twofish	2,576	32.40	6.85x	6.59x	136	11.90	1.41x	1.46x

diagonal line are benchmarks where the optimized implementation is faster.

7.1.4 Experimental Results: Simulation

Although our analysis is conservative in that mitigated code is guaranteed to be leakage free, it is still useful to conduct GEM5 simulations, for two reasons. First, it confirms our analysis reflects the reality: leaks reported by us are real. Second, it demonstrates vividly that after mitigation leaks are indeed eliminated.

Table 7.4 shows our results. For each benchmark, we ran the machine code compiled for x86 on GEM5 using two manually crafted inputs (e.g., cryptographic keys) capable of showing the timing

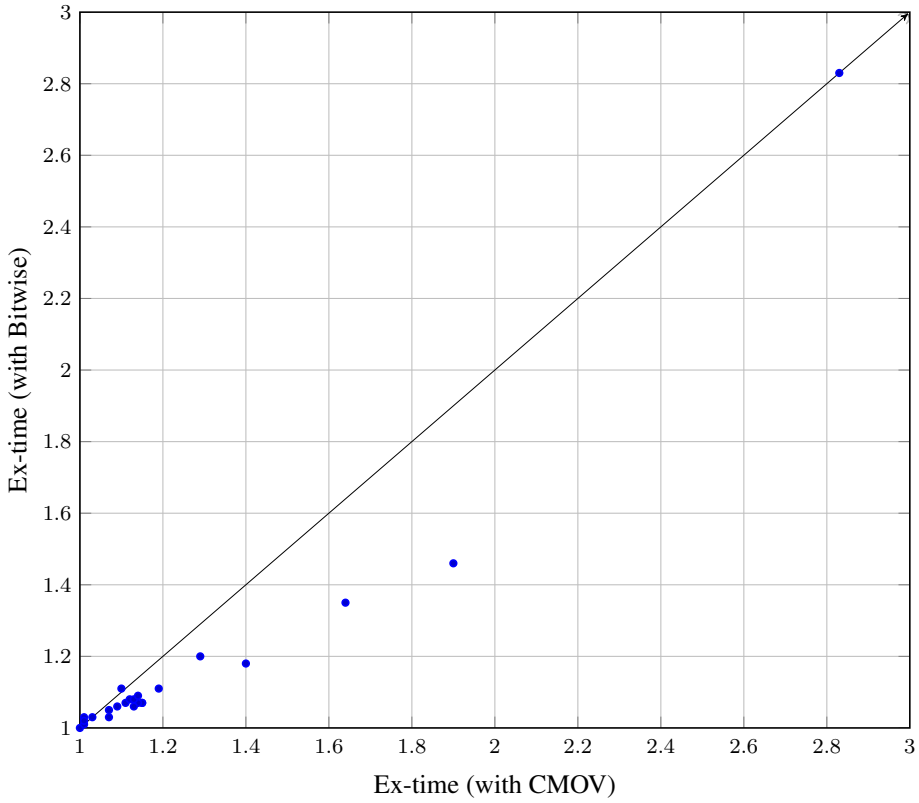


Figure 7.2: *CTSEL* implementations.

variations. Columns 2-5 show the results of the original program, including the number of CPU cycles taken to execute it under the two inputs, as well as the number of cache misses. Columns 6-9 show the results on the mitigated program versions.

The results show the execution time of the original program indeed varies, indicating there are leaks. But it becomes constant after mitigation, indicating leaks are removed. The one exception is *aes_keys*: although it may be a real leak, we were not able to manually craft the inputs under which leak is demonstrable on GEM5. Since the input space is large, manually crafting such inputs is not easy. Perhaps symbolic execution tools can help generate leak-manifesting input pairs — we will consider it for future work.

Table 7.4: Results of GEM5 simulation with 2 random inputs.

Name	Before Mitigation				Mitigation w/o opt		Mitigation w/ opt	
	# CPU cycle (in_1,in_2)		# Miss (in_1,in_2)		# CPU cycle	# Miss	# CPU cycle	# Miss
aes	100,554	101,496	261	269	204,260	303	112,004	303
des	95,630	90,394	254	211	346,170	280	100,694	280
des3	118,362	111,610	271	211	865,656	280	124,176	280
anubis	128,602	127,514	276	275	512,452	276	134,606	276
cast5	102,426	102,070	282	279	266,156	304	108,068	304
cast6	96,992	97,492	238	245	233,774	245	100,914	245
fcrypt	84,616	83,198	224	218	114,576	240	88,236	240
khazad	101,844	100,724	332	322	366,756	432	130,682	432
aes	89,968	90,160	234	235	174,904	240	94,364	240
cast	117,936	117,544	345	342	520,336	436	136,052	435
aes_key*	243,256	243,256	329	329	254,262	329	245,584	328
cast128	161,954	161,694	298	296	305,514	321	167,626	321
des	118,848	119,038	269	270	182,830	317	127,374	316
kasumi	113,362	113,654	204	206	137,914	206	115,060	206
seed	106,518	106,364	239	238	165,546	249	110,486	249
twofish	309,160	299,956	336	334	1,060,832	340	315,018	339
3way	87,834	87,444	181	181	90,844	182	90,844	182
des	152,808	147,344	224	222	181,074	225	168,938	225
loki91	768,064	768,296	181	181	2,170,626	181	2,170,626	181
camellia	84,208	84,020	205	203	102,100	244	91,180	244
des	100,396	100,100	212	211	112,992	213	100,500	213
seed	83,256	83,372	228	230	107,318	240	96,266	239
twofish	230,838	229,948	334	327	982,258	338	295,268	338

7.1.5 Threats to Validity

We now discuss the threats to validity and how they were addressed or could be addressed in future work. First, our mitigation is software-based; as such, we do no address leaks exploitable only by probing the hardware such as instruction pipelines and data buses. We focus on the *Total-time-aware* threat model. Although extensions to handle other threat models are possible (e.g., multi-core and multi-level cache), we consider them as future work.

Although in principle timing characteristics of the machine code may differ from those of the LLVM bit-code, we have taken efforts in making sure machine code produced by our tool does not deviate from the mitigated bit-code. For example, we always align sensitive lookup tables to cache line boundaries, and we implement *CTSEL* as an intrinsic function to ensure constant-time execution.

We also use GEM5 simulation to confirm that machine code produced by our tool is indeed free of timing leaks.

7.2 Cache Analysis under Speculative Execution

We have instantiated our speculative abstract interpretation framework as a speculative cache analysis tool in LLVM [3] and experimentally compared it with a state-of-the-art, non-speculative static cache analysis technique [165]. In our experiments, we used a set-associative cache with the LRU replacement policy, 512 cache lines, and 64 bytes per line. The speculative execution depths, following a cache hit and a cache miss, are set to 20 and 200 instructions, respectively. These bounds were derived from our analysis of the pipelined execution traces produced by GEM5 [29], a state-of-the-art micro-architecture simulator, with *O3CPU*, which is a detailed out-of-order CPU model based on the Alpha 21264 processor.

Our experiments were designed to answer three questions: (1) Is our method more accurate in detecting cache misses than existing methods, which do not consider speculative execution? (2) Is our method fast enough for practical use? (3) Are the optimizations proposed in Section 5.3.2 effective in reducing overhead and increasing accuracy?

7.2.1 Benchmarks

Tables 7.5 and 7.6 show the statistics of our benchmarks, which are collected from various sources including the Malardalen real-time software benchmark [76], a commercially representative embedded software benchmark suite named MiBench [77], a high performance patch for ssh (hpn-ssh) [39], a cryptographic toolkit named *LibTomCrypt* [8], the openssh source code [10], and a Linux kernel for *tegra* [9] used on Tesla automobiles. These benchmarks are divided into two sets: execution time

Table 7.5: Execution time estimation: benchmark statistics.

Name	Source	Description	Loc
adpcm	WCET@mdh	motor control	910
susan	MiBench	image process algorithm	2,140
layer3	MiBench	mp3 audio lib	2,233
jcmarker	MiBench	jpeg compose algorithm	1,444
jdmarker	MiBench	jpeg decompose algorithm	2,068
jcphuff	MiBench	jpeg Huffman entropy encoding routines	694
gtk	MiBench	GTK plotting routines	949
g72	mediaBench	routines for G.721 and G.723 conversions	608
vga	mediaBench	Driver for Borland Graphics Interface	386
stc	mediaBench	pson Stylus-Color Printer-Driver	492

Table 7.6: Side channel detection: benchmark statistics.

Name	Source	Description	Loc
hash	hpn-ssh	hash function	320
encoder	LibTomCrypt	hex encode a string	134
chacha20	LibTomCrypt	chacha20poly1305 cipher	776
ocb	LibTomCrypt	OCB implementation	377
aes	LibTomCrypt	AES implementation	1,838
str2key	openssl	key prepair for des	385
des	openssl	des cipher	1,051
seed	linux-tegra	seed cipher	487
camellia	linux-tegra	camellia cipher	1,324
salsa	linux-tegra	Salsa20 stream cipher	279

estimation and side channel detection. The benchmarks for execution time estimation (Table 7.5) are used as is, whereas the benchmarks for side channel detection (Table 7.6) are used together with a client program that we wrote, to invoke the benchmark program in a way similar to Figure 4.7.

7.2.2 Effectiveness: Execution Time Estimation

We first compare our method with the state-of-the-art, non-speculative method [165]. The results are shown in Table 7.7. For our method, we also report the number of speculative cache misses (#SpMiss), which are not observable from outside of the CPU, the number of conditional branches

Table 7.7: Execution time estimation: comparisons in terms of the analysis time and the number of cache misses.

Name	Non-speculative		Speculative				
	Time (s)	#Miss	Time (s)	#Miss	#SpMiss	#Branch	#Iteration
adpcm	0.98	24	12.70	32	17	75	173
susan	19.40	17	248.40	26	17	113	464
layer3	7.24	78	65.54	88	35	241	374
jcmarker	0.20	22	3.40	26	11	37	72
jdmarker	2.89	21	15.18	78	55	193	726
jcphuff	0.03	12	0.44	12	13	25	32
gtk	19.90	16	274.76	19	13	77	190
g72	0.16	6	0.94	9	4	41	79
vga	0.05	4	0.06	4	3	3	3
stc	0.13	10	0.96	23	14	39	105

that can be speculatively executed, and the total number of iterations of our method on loops.

The results show that our method detected more cache misses, thus highlighting the unsoundness of the existing method and the importance of modeling speculative execution during execution time estimation.

As for the analysis time, our method completed all the benchmarks, although it took a longer time than the non-speculative analysis due to its focus on being always sound. The reason why it took significantly longer for the *gtk* benchmark, in particular, is because the program has a large data size (of nearly 3 MB), which led to a large number of variables to be tracked in the abstract cache state.

Table 7.8 compares two merging strategies in terms of the analysis time, the number of cache misses, the number of speculative cache misses, and the number of iterations. The result is somewhat surprising in that although *merging at rollback point* is more aggressive than *just-in-time merging*, the later is actually faster while being more accurate. The reason is because merging the speculative state with the normal state right after the rollback point may force the normal state to become a coarser-grained over-approximation. This can lead to a slower convergence to a coarser fixed point, as shown by the data in Columns 5 and 9. However, there are exceptions, indicating that *optimal merging* in general is *problem-specific*, and the accuracy depends on the combined effects

Table 7.8: Execution time estimation: comparisons of two strategies for merging speculative executions.

Name	Merging at rollback point				Just-in-time merging			
	Time(s)	#Miss	#SpMiss	#Ite	Time(s)	#Miss	#SpMiss	#Ite
adpcm	14.40	31	25	261	12.70	32	17	173
susan	405.70	30	29	620	248.40	26	17	464
layer3	84.64	94	53	471	65.54	88	35	374
jcmarker	4.80	27	19	99	3.40	26	11	72
jdmarker	16.11	35	59	777	15.18	78	55	726
jcphuff	0.48	12	10	36	0.44	12	13	32
gtk	358.56	24	26	236	274.76	19	13	190
g72	1.28	7	1	122	0.94	9	4	79
vga	0.07	4	3	5	0.06	4	3	3
stc	1.86	31	35	222	0.96	23	14	105

of branches and loops in a program.

7.2.3 Effectiveness: Side Channel Detection

Table 7.9 shows the results for side channel detection, including comparisons of the two methods in terms of the analysis time and whether leaks are detected. In this context, a leak refers to the dependency between the cache behavioral difference and sensitive data; furthermore, whether there is a leak or not often depends on the input buffer size controlled by the (potentially malicious) user. Thus, during experiments, we set the buffer size to various values from 32K bytes (the size of cache we use) down to 0 byte.

Generally speaking, the larger the buffer size, the easier that the client program triggers the behavioral difference. Thus we first set the buffer size to 32KB, and starting from there we gradually reduce the buffer size and keep track of the impact of speculative execution on cache state, until the two methods return different results.

Since the benchmarks are mostly cryptographic algorithms, which are relatively small in terms of the number of lines of code, the analysis time is short. Furthermore, our method successfully

Table 7.9: Side channel detection: comparisons in terms of the analysis time and whether leaks are detected.

Name	Buffer (byte)	Traditional		Speculative	
		Time (s)	Leak Detected	Time (s)	Leak Detected
hash	31,808	0.67	No	1.15	Yes
encoder	32,512	0.03	No	0.10	Yes
chacha20	26,304	1.18	No	9.24	Yes
ocb	31,616	0.10	No	0.68	Yes
aes	32,768	0.08	No	2.13	No
str2key	32,768	0.01	No	0.01	No
des	0	0.60	No	14.20	Yes
seed	32,768	0.01	No	0.07	No
camellia	32,768	0.35	No	6.35	No
salsa	32,768	0.02	No	0.06	No

detected leaks in half of the benchmarks, whereas the existing (unsound) method did not detect leaks in any of them. This highlights the importance of having a sound static cache analysis for speculative executions, e.g., to detect more leaks and avoid producing bogus proofs (that there is no leak). On one of the benchmarks, *des*, leaks are detected even if the buffer size is set to 0; this is because, even without the client program, the benchmark program itself has a user controlled buffer, which can be set to sizes that induce timing side-channel leaks under speculative execution.

As a static analysis procedure, our method may generate false positives. In addition to abstraction, the other source of false positives is modeling of the speculative execution. For each of the new leaks detected by our method in Table 7.9, we have manually inspected the software code and the execution trace. Our inspecction confirmed that all of them are actually real; that is, there exist specific memory/cache layouts and execution traces that induce the leaks.

7.3 Boolean Shield under Burst Error

Our shield synthesis approach for Boolean domain is implemented in the same software tool that also implements the method of Bloem et al. [31]. The fix-point computation for solving

safety games is implemented symbolically, using CUDD [4] as the BDD library, whereas the construction of the various monitors and the game graph are carried out explicitly. The tool takes the automaton representation of the safety specification φ^s as input and returns the Verilog program of the synthesized shield $\mathcal{S}$ as output.

7.3.1 Benchmarks

We have evaluated our method on a range of safety specifications, including temporal logic properties from (1) the Toyota powertrain control verification benchmark [83], (2) an automotive design for engine and brake controls [117], (3) the traffic light controller example from the VIS model checker [34], (4) LTL property specification patterns from Dwyer et al. [52], and (5) parts of the ARM AMBA bus arbiter specification [30].

Specifically, properties from [83] are on the model of a fuel control system, specifying the performance requirements in various operation modes. Originally, they were represented in signal temporal logic (STL). We translated them to LTL by replacing the predicates over real variables with boolean variables. The properties for engine and brake control [117] are related to the safety of the brake overriding mechanism. The properties for traffic light controller [34] are for safety of a crossroads traffic light. The AMBA benchmark [30] includes combinations of various properties of an ARM bus arbiter. We also translate liveness properties in Dwyer et al. [52] to safety properties by adding a bound on the reaction time steps. For example, in the first columns of Table 7.11, the numbers besides F and U are the bound number, where F and U mean *Finally* and *Until* respectively.

7.3.2 Experimental Results

Table 7.10 shows the results of running our tool on these benchmarks and comparing it with the method of Bloem et al. [31]. Columns 1-2 show the properties we use from [30] and the number

Table 7.10: Experimental results for comparing the two shield synthesis algorithms.

Property φ^s	Sates	K-Stabilizing Shield			Burst-Error Shield		
		Handle-Burst-Error	States in $\mathcal{S}$	Time (s)	Handle-Burst-Error	States in $\mathcal{S}$	Time (s)
AMBA G1+2+3	12	yes	22	0.1	yes	22	0.1
AMBA G1+2+4	8	no (1-step)	61	6.3	yes	78	2.2
AMBA G1+3+4	15	no (1-step)	231	55.6	yes	640	97.6
AMBA G1+2+3+5	18	no (1-step)	370	191.8	yes	1,405	61.8
AMBA G1+2+4+5	12	no (1-step)	101	3,992.9	yes	253	472.9
AMBA G4+5+6	26	no (2-step)	252	117.9	yes	205	26.4
AMBA G5+6+10	31	no (2-step)	329	9.8	yes	396	31.4
AMBA G5+6+9e4+10	50	no (2-step)	455	17.6	yes	804	42.1
AMBA G5+6+9e8+10	68	no (2-step)	739	34.9	yes	1,349	86.8
AMBA G5+6+9e16+10	104	no (2-step)	1,293	74.7	yes	2,420	189.7
AMBA G5+6+9e64+10	320	no (2-step)	4,648	1,080.8	yes	9,174	2,182.5
AMBA G8+9e4+10	48	no (2-step)	204	7.0	yes	254	6.1
AMBA G8+9e8+10	84	no (2-step)	422	22.5	yes	685	33.7
AMBA G8+9e16+10	156	no (2-step)	830	83.7	yes	1,736	103.1
AMBA G8+9e64+10	588	no (2-step)	3,278	2,274.2	yes	7,859	2,271.5

of states of the safety specification φ^s. Columns 3-5 show the results of applying the k-stabilizing shield synthesis algorithm [31], including whether the resulting shield can handle burst error, the shield size in terms of the number of states, and the synthesis time in seconds. Similarly, Columns 6-8 show the results of applying our new synthesis algorithm. Note that the k-stabilizing shields are guaranteed to handle burst error, and as shown in Table 7.10, only some of them can actually handle burst error. Here, "*no (1-step)*" means the shield needs at least one more clock cycle to recover from the previous error before it can take on the next error, and "*no (2-step)*" means the shield needs at least two more clock cycles to recover. In contrast, the shield synthesized by our new method can recover instantaneously and therefore can always handle burst error.

In terms of the synthesis time, the result is mixed in that our new method is sometimes slower and sometimes faster than the existing method. There are two reasons for such results. On the one hand, our method is searching through a significantly larger game graph than the existing method in order to find the best winning strategy for handling burst error. On the other hand, our method utilizes the new optimization technique described in Section 5.3.2 for symbolically computing the winning region, which can significantly speed up the fix-point computation.

Table 7.11: Experimental results for synthesizing the shield with and without optimization.

Property φ^s	States	Burst Error Shield Syn. (w/o Opt)		Burst Error Shield Syn. (w/ Opt)	
		States in $\mathcal{S}$	Time (s)	States in $\mathcal{S}$	Time (s)
Toyota powertrain	23	38	0.3	38	0.3
Engine and brake ctrl	5	7	0.1	7	0.1
Traffic light	4	7	0.2	7	0.2
$\mathsf{F}_{256}\, p$	259	259	45.5	259	10.5
$\mathsf{F}_{512}\, p$	515	5157	511.0	515	54.4
$\mathsf{G}(\neg q) \vee \mathsf{F}_{64}(q \wedge \mathsf{F}_{64}\, p)$	67	67	0.8	67	0.6
$\mathsf{G}(\neg q) \vee \mathsf{F}_{256}(q \wedge \mathsf{F}_{256}\, p)$	259	259	46.2	259	10.7
$\mathsf{G}(\neg q) \vee \mathsf{F}_{512}(q \wedge \mathsf{F}_{512}\, p)$	515	515	668.1	515	54.5
$\mathsf{G}(q \wedge \neg r \rightarrow (\neg r\, \mathsf{U}_8(p \wedge \neg r)))$	10	4,002	3.9	5,519	4.5
$\mathsf{G}(q \wedge \neg r \rightarrow (\neg r\, \mathsf{U}_{12}(p \wedge \neg r)))$	14	95,357	1,506.9	27,338	1,414.5
AMBA G1+2+4	8	69	2.3	78	2.2
AMBA G1+3+4	15	566	99.5	640	97.6
AMBA G1+2+3+5	18	1,256	58.4	1,405	61.8
AMBA G1+2+4+5	12	193	479.2	253	472.9
AMBA G4+5+6	26	206	26.3	205	26.4
AMBA G5+6+9e16+10	104	2,334	194.2	2,420	189.7
AMBA G5+6+9e64+10	320	8,618	2,865.6	9,174	2,182.5
AMBA G8+9e16+10	156	1,344	111.0	1,736	103.1
AMBA G8+9e64+10	588	5,848	7,843	7,859	2,271.5

Table 7.11 shows the results of our synthesis algorithm with and without optimization. Columns 1-2 show the benchmark name and the size of the safety specification. Columns 3-4 show the size of the resulting shield and the synthesis time without using the optimization. Columns 5-6 show the shield size and the synthesis time with the optimization. In almost all cases, there is significant reduction in the synthesis time when the optimization is used. At the same time, there is slightly difference in the number of states in the resulting shield. This is because the game graph often contains multiple winning strategies, and currently our method for computing the winning strategy tends to pick an arbitrary one. Furthermore, since the shield is implemented in hardware, the difference in the number of bit-registers (flip-flops) needed to implement the two shields will be further reduced. For example, in the last benchmark, we have $\lceil log_2(3278) \rceil = 12$, whereas $\lceil log_2(7859) \rceil = 13$, meaning that the shield requires either 12 or 13 bit-registers. Nevertheless, for future work, we plan to investigate new ways of computing the winning strategy to further reduce the shield size.

7.4 Real-Valued Shield Synthesis

The shield synthesis approach for real domain has been implemented as a tool that takes the automaton representation of a safety specification as input and returns a real-valued shield as output. Internally, we solve the safety game using Mazala's algorithm [109] implemented symbolically using CUDD [4], and use the LP solver integrated in Z3 [47] for prediction, validation and constraint solving. For evaluation purposes, the shield is implemented as a C program and is executed at every time step. Each execution has two phases: (1) generating Boolean values for signals in O', and (2) generating real values for signals in O'_r.

7.4.1 Benchmarks

We evaluated our tool on seven sets of benchmarks, including automotive powertrain control [83], autonomous driving [132], adaptive cruise control [120], multi-drone fleet control [123], generic control [84], blood glucose control [138], and water tank control [16]. In all benchmarks, the original specification was given in STL, which has both timing and real-valued constraints.

Table 7.12 shows the benchmark statistics, including the application name, the property, a short description, and the corresponding STL formula. For brevity, we omit the automaton representations, but they will be released together with our tool upon acceptance of the paper. We conducted experiments on a computer with Intel i5 3.1GHz CPU, 8GB RAM, and the Ubuntu 14.04 operating system. Our experiments were designed to answer the following questions: (1) Is our tool efficient in synthesizing the real-valued shield? (2) Is the shield effective in preventing safety violations? (3) Are the real-valued signals produced by the shield of high quality?

Table 7.12: Statistics of the benchmark applications.

Application	Property	STL Formula and Description
Powertrain	R26	In normal mode, permitted overshoot/undershoot is always be less than 0.05 $\mathsf{G}_{[\tau_s,T]}\big(l = \mathsf{normal} \Rightarrow \lvert\mu\rvert < 0.05\big)$
	R27	In normal mode, overshoot/undershoot less than 0.02 within the settling time $\mathsf{G}_{[\tau_s,T]}\Big(\mathsf{rise}(a) \lvert \mathsf{fall}(a) \Rightarrow \mathsf{G}_{[\eta,\frac{\varsigma}{2}]}\big(\lvert\mu\rvert < 0.02\big)\Big)$
	R32	From power to normal, overshoot/undershoot less than 0.02 within settling time $\mathsf{G}_{[\tau_s,T]}\Big(l = \mathsf{power} \wedge \mathsf{X}(l = \mathsf{normal}) \Rightarrow \mathsf{G}_{[\eta,\frac{\varsigma}{2}]}\big(\lvert\mu\rvert < 0.02\big)\Big)$
	R33	In power mode, permitted overshoot or undershoot should be less than 0.2 $\mathsf{G}_{[\tau_s,T]}\big(l = \mathsf{power} \Rightarrow \lvert\mu\rvert < 0.2\big)$
	R34	Upon startup/sensor failure, overshoot/undershoot <0.1 within the settling time $\mathsf{G}_{[\tau_s,T]} l = \mathsf{startup} \lvert \mathsf{sensor\_fail} \wedge \mathsf{rise}(a) \lvert \mathsf{fall}(a) \Rightarrow \mathsf{G}_{[\eta,\frac{\varsigma}{2}]}\big(\lvert\mu\rvert < 0.1\big)\Big)$
Autonomous Driving	D1	Vehicle should keep a steady speed V_s when there is no collision risk $\mathsf{G}\big(\lvert y_k^{ego} - x_k^{adv}\rvert >= 4\big) \Rightarrow \mathsf{G}\big(\lvert v_k^{ego} - V_s\rvert < \varepsilon\big)$
	D2	Vehicle should come to stop for at least 2 second when there is collision risk $\mathsf{G}\big(\lvert y_k^{ego} - x_k^{adv}\rvert < 4\big) \Rightarrow \mathsf{G}_{[0,2]}\big(\lvert v_k^{ego}\rvert < 0.1\big)$
Cruise Control	A1	Keep a safe distance with lead vehicle: $\mathsf{G}\big(\mathsf{pos\_lead}[t] - \mathsf{pos\_ego}[t] > D_s\big)$
	A2	Achieve cruise velocity if there is a comfortable distance $\big(\mathsf{pos\_lead}[t] - \mathsf{pos\_ego}[t] > D_c\big)\mathsf{U}_{[0,10]}\big(\lvert\mathsf{v\_ego}[t] - \mathsf{v\_cruise}[t]\rvert < \varepsilon\big)$
	A3	Vehicle should never travel backward: $\mathsf{G}\big(\mathsf{v\_ego}[t] >= 0\big)$
	A4	Vehicle doesn't halt unless lead vehicle halts: $\mathsf{G}\big(\mathsf{v\_lead}[t] > 0\big) \Rightarrow \mathsf{G}\big(\mathsf{v\_ego}[t] > 0\big)$
Quadrotor Control	Q1	Drone flies to goal point if no obstacles are on they way: $\mathsf{G}\big(Obs(\mathbf{pos}^{quad}, \mathbf{pos}^{obs}) \Rightarrow \omega_g > 0\big)\big)$
	Q2	Avoiding obstacles:$\mathsf{G}\neg Obs(\mathbf{pos}^{quad}, \mathbf{pos}^{obs}) \Rightarrow$ $\Big(\omega_{\vec{g}} > 0 \wedge \mathsf{G}\big(Dis(\mathbf{pos}^{quad}, \mathbf{pos}^{obs}) < \varepsilon \Rightarrow \omega_g = 0\big)\Big)$
General Control	C1	After settling, output error should be less than set value ε_b: $\mathsf{G}\Big(\mathsf{x}[t] \Rightarrow \mathsf{G}_{[10,\infty]}\big(\lvert\frac{\mathsf{y}[t]-\mathsf{y}^{ref}}{\mathsf{y}^{ref}}\rvert < \varepsilon_b\big)\Big)$
	C2	Output error should be $[\varepsilon^{\perp}, \varepsilon^{\top}]$ in settling time: $\mathsf{G}\Big(\mathsf{x}[t] \Rightarrow \mathsf{G}_{[0,20]}\big(\varepsilon^{\perp} < \frac{\mathsf{y}[t]-\mathsf{y}^{ref}}{\mathsf{y}^{ref}} < \varepsilon^{\top}\big)\Big)$
	C3	Output should achieve reference value within $rise\_time$: $\mathsf{G}\Big(\mathsf{x}[t] \Rightarrow \mathsf{F}_{[0,\mathsf{rise\_time}]}\big(\lvert\frac{\mathsf{y}[t]-\mathsf{y}^{ref}}{\mathsf{y}^{ref}}\rvert < \varepsilon_r\big)\Big)$
Glucose Control	B1	Having meal within t_1 minutes after taking the bolus is safe. A bolus must be taken after t_2 minutes of having meal, if it is not yet taken: $\mathsf{G}\Big(\mathsf{F}_{[0,t_1+t_2]}(B > \mathsf{c}_2) \vee \mathsf{G}_{[t_1,t_1+t_2]}\big(M > \mathsf{c}_1 \Rightarrow \mathsf{F}_{[0,t_2]}(B > \mathsf{c}_2)\big)\Big)$
Water Tank Control	W1	Turn on inflow and turn off outflow switch when water level is low ($l < 4$) $\mathsf{G}\big(l < 4 \Rightarrow \mathsf{G}_{[0,3]}(\mathsf{flow}_{out} = 0 \wedge 1 < \mathsf{flow}_{in} < 2)\big)$
	W2	Turn on outflow and turn off inflow switch when water level is high ($l > 93$) $\mathsf{G}\big(l > 93 \Rightarrow \mathsf{G}_{[0,3]}(\mathsf{flow}_{in} = 0 \wedge 0 < \mathsf{flow}_{out} < 1)\big)$

7.4.2 Experimental Results

Table 7.13 shows the results of our shield synthesis procedure. Columns 1-3 show the property name, the number of states of the specification, and the number of real-valued signals in I_r and O_r, respectively. Column 4 shows the number of predicates defined over signals in I_r and O_r. Based on these predicates, Boolean signals in I and O are created; Column 5 shows the number

Table 7.13: Results of real shield synthesis procedure.

Name	Specification		Synthesis Tool				Shield $\mathcal{S}$	
	states	$\|I_r\|/\|O_r\|$	$\|\mathcal{P}_I\|/\|\mathcal{P}_O\|$	$\|I\|/\|O\|$	$\|\mathcal{R}\|/\|\mathcal{F}\|$	time(s)	states	constrs
R26+R27	8	1/1	2/2	5/2	2/1	0.16	25	2+2
R32+R33	9	1/1	2/2	5/2	2/1	0.15	28	2+2
R26+R27+R32 +R33+R34	23	1/1	2/4	5/4	12/11	1.15	158	4+2
D1	6	3/1	5/3	6/3	53/5	0.15	19	3+2
D2	5	3/1	2/3	3/3	5/5	0.21	30	3+2
D1+D2	14	3/1	5/3	6/3	53/5	0.8	164	3+2
A1+A3+A4	3	3/1	2/2	2/3	1/1	0.08	8	2+0
A2+A3+A4	4	4/1	3/3	3/3	4/4	0.1	15	3+0
A1+A2+A3+A4	7	4/1	4/3	4/4	8/4	0.55	48	3+0
Q1+Q2	5	1/2	1/2	2/2	0/0	0.08	7	2+0
C1+C2+C3	19	2/1	3/4	3/4	13/11	0.52	118	4+2
B1	5	3/1	5/1	5/1	14/0	0.1	6	1+0
W1+W2	6	1/2	2/2	2/2	1/0	0.1	10	2+2

of these signals. Column 6 shows the number of conflicting constraints captured by the relaxation and feasibility automata, respectively. Column 7 shows the synthesis time. Columns 8-9 show the number of states of the Boolean shield, and the number of real-valued constraints to be solved at run time.

Table 7.14 shows the runtime performance of the shields. For each shield, we generated input signals (for I_r and O_r) based on the description of the system: some of these input signals satisfy the specification while others do not. By measuring the response time of the shield under these input signals, as well as the quality of the corrections made by the shield, we hope to evaluate its effectiveness.

In this table, Column 1 shows the property name. Column 2 shows the size of the C program that implements the shield. Column 3 shows the response time of the Boolean shield on input signals that do not violate the specification. Columns 4-5 show the response time on input signals that violate the specification. Among these columns, *prediction* means the real-valued solution was successfully computed by a linear regression, whereas *constr. solving* means prediction failed and the solution was computed by the LP solver.

Table 7.14: Results of evaluating runtime performance of the shield.

Name	Implementation (LoC)	Shield Response Time		
		Boolean step (us)	prediction step (us)	constraint solving (us)
R26+R27	745	0.3	293.3	336.8
R32+R33	748	0.41	256.5	333.9
R26+R27+R32 +R33+R34	1446	0.8	245.0	279.8
D1	781	0.45	177.2	164.7
D2	853	0.5	313.5	329.0
D1+D2	2242	0.8	318.3	202.4
A1+A3+A4	539	0.37	164.3	212.7
A2+A3+A4	632	0.49	281.7	431.5
A1+A2+A3+A4	940	0.45	291.7	290.1
Q1+Q2	556	0.18	299.2	313.5
C1+C2+C3	1037	0.5	299.4	395.2
B1	623	0.31	225.4	313.4
W1+W2	608	0.57	295.3	222.1

Overall, the time to compute real-valued correction signals is within 0.5 ms when $\mathcal{D} \not\models \varphi$, and less than 1 us when $\mathcal{D} \models \varphi$. In the latter case, the shield does not need to make correction at all. In both cases, the response time is always bounded and fast enough for the target applications.

7.4.3 Case Studies

Case Study 1: Powertrain Control System To validate the effectiveness of our approach, we integrated the shield into the simulation model of the powertrain control system. Then, we compared the system performance with and without the shield. Fig. 7.3 shows the simulation results, where our shield was synthesized from the system properties 26, 27, 32, 33 and 34 as described in Jin et al. [83]. Recall that μ is the normalized error of the A/F ratio and μ_{ref} is a reference value.

The green dashed line indicates the safe region, which varies as the system switches between different modes (transition events are highlighted with black dotted line). The red dashed line represents violations of the specification by the original O_r signals. The solid red line represents corrections made in O'_r. The result shows that our shield can always produce real-valued correction

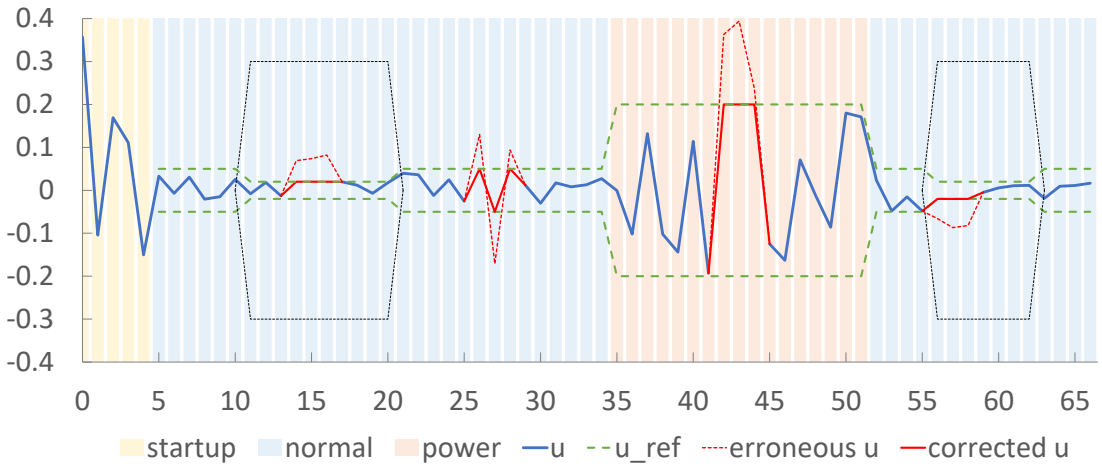


Figure 7.3: Automotive powertrain system simulation results (with and without the shield).

to keep μ in the safe region.

Case Study 2: Autonomous Driving

Fig. 7.4 shows the simulation results of an autonomous driving system [132] with and without our shield. Here, an ego vehicle is put into a nondeterministic environment that includes an adversarial vehicle, and the two cars are crossing an intersection. The ego vehicle is protected by a shield synthesized from D1+D2 in Table 7.12. The three plots, from top to bottom, are for distances to the intersection, velocities, and accelerations of the two vehicles. The x-axis represents the time in seconds.

The adversarial vehicle drives straight through the intersection at a constant speed. The ego vehicle, in contrast, may change speed to avoid collision. From $t = 0s$ to $t = 5s$, since the distance between the two vehicles is large, the ego vehicle maintains a steady speed (set to $2m/s$ initially). At $t = 5s$, based on the safety specification, it is supposed to come to a stop (for at least $2s$ or when there is no collision risk). However, since we injected an error at $t = 6s$ (in red dashed line), there is an unexpected acceleration and, without the shield, there would have been a collision.

The blue lines show the behavior of the ego vehicle after corrections are made by our shield. Clearly,

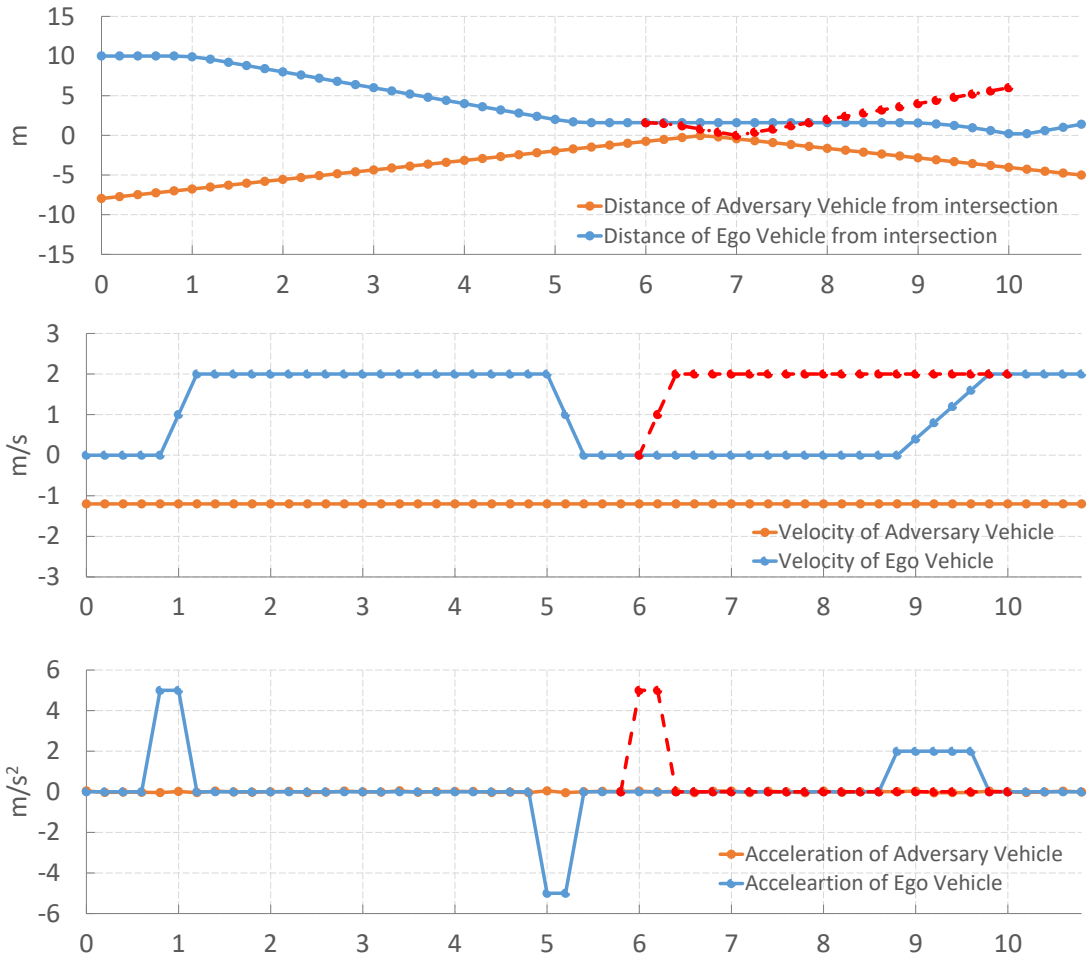


Figure 7.4: Position, velocity and acceleration in autonomous driving simulation.

its behavior satisfies the requirements: it stops at the intersection to allow the adversarial vehicle to pass safely. Furthermore, the real-valued correction made by our shield is successfully predicted using linear regression, and the predicted values satisfy not only the safety but also the robustness requirements.

Chapter 8

Conclusions

This dissertation presents two types of techniques for enforcing the safety and security of critical software systems.

In terms of security, we have presented a number of techniques for detecting timing side-channel leaks via static program analysis, and a number of techniques for mitigating these leaks via program transformation. Toward this end, we have lifted the abstract interpretation framework to make it sound for handling non-functional properties such as cache timing and micro-architectural optimizations such as speculative execution. These proposed techniques have been implemented in software tools based on the LLVM compiler and evaluated on realistic applications. Our experimental results show that the techniques are both effective and efficient in practice.

In terms of safety, we have presented a number of techniques for synthesizing a runtime enforcer, called a shield, to instantaneously generate correction signals. The shields synthesized by our techniques not only handle Boolean signals, but also handle real-valued signals. Furthermore, they can robustly handle burst error. These proposed techniques have also been implemented in software tools and evaluated on realistic embedded control systems. Our experimental results show that the

shields produced by our techniques are both effective and efficient in enforcing safety properties of reactive systems.

Bibliography

[1] *Botan: Crypto and TLS for C++11*. https://github.com/randombit/botan/.

[2] *Fair Evaluation of Lightweight Cryptographic Systems*. https://www.cryptolux.org/index.php/FELICS.

[3] *The LLVM Compiler Infrastructure*. http://llvm.org/.

[4] *CUDD: CU Decision Diagram Package*. ftp://vlsi.colorado.edu/pub/.

[5] *System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives*. https://bench.cr.yp.to/supercop.html.

[6] *Libgcrypt*. https://www.gnupg.org/software/libgcrypt/index.html.

[7] *Intel ®64 and IA-32 Architectures Optimization Reference Manual*. https://botan.randombit.net/, 2014.

[8] *LibTomCrypt: A Modular and Portable Cryptographic Toolkit*. https://www.libtom.net/LibTomCrypt, 2018.

[9] *Tesla Motors: Linux*. https://github.com/teslamotors/linux, 2018.

[10] Openssh, 2018. URL http://www.openssh.com/.

[11] Onur Aciiçmez. Yet another microarchitectural attack:: exploiting i-cache. In *Proceedings of the 2007 ACM workshop on Computer security architecture*, pages 11–18. ACM, 2007.

[12] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 312–320. ACM, 2007.

[13] Johan Agat. Transforming out timing leaks. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 40–53, 2000.

[14] Giovanni Agosta, Alessandro Barenghi, and Gerardo Pelosi. A code morphing methodology to automate power analysis countermeasures. In *ACM/IEEE Design Automation Conference*, pages 77–82, 2012.

[15] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE Symposium on Security and Privacy*, pages 526–540, 2013.

[16] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18)*, pages 2669–2678, 2018.

[17] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[18] Mário S. Alvim, Konstantinos Chatzikokolakis, Annabelle McIver, Carroll Morgan, Catuscia Palamidessi, and Geoffrey Smith. Additive and multiplicative notions of leakage, and their capacities. In *IEEE Computer Security Foundations Symposium*, pages 308–322, 2014.

[19] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. Decomposition instead of self-composition for proving the absence of timing chan-

nels. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 362–375, 2017.

[20] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In *ACM Conference on Computer and Communications Security*, pages 297–307, 2010.

[21] Michael Backes and Boris Köpf. Formally bounding the side-channel leakage in unknown-message attacks. In *European Symposium on Research in Computer Security*, pages 517–532, 2008.

[22] George Balatsouras and Yannis Smaragdakis. Structure-sensitive points-to analysis for C and C++. In *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, pages 84–104, 2016. doi: 10.1007/978-3-662-53413-7_5. URL https://doi.org/10.1007/978-3-662-53413-7_5.

[23] Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Pasareanu, and Tevfik Bultan. String analysis for side channels with segmented oracles. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 193–204, 2016.

[24] Gilles Barthe, Tamara Rezk, and Martijn Warnier. Preventing timing leaks through transactional branching instructions. *Electr. Notes Theor. Comput. Sci.*, 153(2):33–55, 2006.

[25] Gilles Barthe, Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Leakage resilience against concurrent cache attacks. In *International Conference on Principles of Security and Trust*, pages 140–158, 2014.

[26] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, 2011. ISSN 1049-331X.

[27] Ali Galip Bayrak, Francesco Regazzoni, Philip Brisk, François-Xavier Standaert, and Paolo Ienne. A first step towards automatic application of power analysis countermeasures. In *ACM/IEEE Design Automation Conference*, pages 230–235, 2011.

[28] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *International Cryptology Conference*, pages 513–525, 1997.

[29] Nathan L. Binkert, Bradford M. Beckmann, Gabriel Black, Steven K. Reinhardt, Ali G. Saidi, Arkaprava Basu, Joel Hestness, Derek Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib Bin Altaf, Nilay Vaish, Mark D. Hill, and David A. Wood. The GEM5 simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[30] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. *J. Comput. Syst. Sci.*, 78(3):911–938, 2012.

[31] R. Bloem, B. Könighofer, R. Könighofer, and C. Wang. Shield synthesis: Runtime enforcement for reactive systems. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2015.

[32] Borzoo Bonakdarpour and Bernd Finkbeiner. The complexity of monitoring hyperproperties. In *IEEE Computer Security Foundations Symposium*, pages 162–174, 2018.

[33] Benjamin A. Braun, Suman Jana, and Dan Boneh. Robust and efficient elimination of cache and timing side channels. *CoRR*, abs/1506.00189, 2015. URL http://arxiv.org/abs/1506.00189.

[34] R. K. Brayton et al. VIS: A system for verification and synthesis. In *International Conference on Computer Aided Verification*, pages 428–432, 1996.

[35] Tegan Brennan, Seemanta Saha, and Tevfik Bultan. Symbolic path cost analysis for side-

channel detection. In *International Conference on Software Engineering*, pages 424–425, 2018.

[36] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[37] Tevfik Bultan, Fang Yu, Muath Alkhalaf, and Abdulbaki Aydin. *String Analysis for Software Verification and Security*. 2017.

[38] Jia Chen, Yu Feng, and Isil Dillig. Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 875–890, 2017.

[39] Rapier Chris, Steven Michael, Bennett Benjamin, and Tasota Mike. High performance ssh/scp - hpn-ssh, 2018 (accessed March 1, 2019). URL https://www.psc.edu/hpn-ssh.

[40] Duc-Hiep Chu, Joxan Jaffar, and Rasool Maghareh. Precise cache timing analysis via symbolic execution. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 1–12, 2016.

[41] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logics of Programs*, LNCS 131, pages 52–71, 1981.

[42] David Cock, Qian Ge, Toby C. Murray, and Gernot Heiser. The last mile: An empirical study of timing channels on seL4. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 570–581, 2014.

[43] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *IEEE Symposium on Security and Privacy*, pages 45–60, 2009.

[44] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[45] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 84–96, 1978.

[46] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Thwarting cache side-channel attacks through dynamic software diversity. In *Annual Network and Distributed System Security Symposium*, 2015.

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

[48] Matthew Dellinger, Piyush Garyali, and Binoy Ravindran. Chronos linux: a best-effort real-time multiprocessor linux kernel. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 474–479. IEEE, 2011.

[49] Daniel Dinu, Yann Le Corre, Dmitry Khovratovich, Léo Perrin, Johann Großschädl, and Alex Biryukov. Triathlon of lightweight block ciphers for the internet of things. Cryptology ePrint Archive, Report 2015/209, 2015. http://eprint.iacr.org/2015/209.

[50] Adel Dokhanchi, Bardh Hoxha, and Georgios E. Fainekos. On-line monitoring for temporal logic robustness. In *International Conference on Runtime Verification*, pages 231–246, 2014.

[51] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A tool for the static analysis of cache side channels. In *USENIX Security*, pages 431–446, 2013.

[52] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *International Conference on Software Engineering*, 1999.

[53] R. Ehlers and U. Topcu. Resilience to intermittent assumption violations in reactive synthesis. In *HSCC*, pages 203–212. ACM, 2014.

[54] Hassan Eldib and Chao Wang. An SMT based method for optimizing arithmetic computations in embedded software code. In *International Conference on Formal Methods in Computer-Aided Design*, 2013.

[55] Hassan Eldib and Chao Wang. Synthesis of masking countermeasures against side channel attacks. In *International Conference on Computer Aided Verification*, pages 114–130, 2014.

[56] Hassan Eldib and Chao Wang. An SMT based method for optimizing arithmetic computations in embedded software code. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 33(11): 1611–1622, 2014.

[57] Hassan Eldib, Chao Wang, and Patrick Schaumont. SMT based verification of software countermeasures against side-channel attacks. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2014.

[58] Hassan Eldib, Chao Wang, and Patrick Schaumont. Formal verification of software countermeasures against side-channel attacks. *ACM Trans. Softw. Eng. Methodol.*, 24(2):11:1–11:24, 2014.

[59] Hassan Eldib, Chao Wang, Mostafa Taha, and Patrick Schaumont. QMS: Evaluating the side-channel resistance of masked software from source code. In *ACM/IEEE Design Automation Conference*, pages 209:1–6, 2014.

[60] Hassan Eldib, Chao Wang, Mostafa Taha, and Patrick Schaumont. Quantitative Masking

Strength: Quantifying the side-channel resistance of masked software code. In *IEEE Trans. on CAD of Integrated Circuits and Systems*, volume 34, pages 1558–1568, 2015.

[61] Hassan Eldib, Meng Wu, and Chao Wang. Synthesis of fault-attack countermeasures for cryptographic circuits. In *International Conference on Computer Aided Verification*, pages 343–363, 2016.

[62] Georgios E. Fainekos and George J. Pappas. Robustness of temporal logic specifications. In *Formal Approaches to Software Testing and Runtime Verification, First Combined International Workshops, FATES 2006 and RV 2006, Seattle, WA, USA, August 15-16, 2006, Revised Selected Papers*, pages 178–192, 2006.

[63] Y. Falcone, J.-C. Fernandez, and L. Mounier. What can you verify and enforce at runtime? *STTT*, 14(3):349–382, 2012.

[64] Samira S Farahani, Vasumathi Raman, and Richard M Murray. Robust model predictive control for signal temporal logic synthesis.

[65] Christian Ferdinand and Reinhard Wilhelm. On predicting data cache behavior for real-time systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 16–30, 1998.

[66] Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2-3):131–181, 1999.

[67] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache behavior prediction by abstract interpretation. *Science of Computer Programming*, 35(2-3):163–189, 1999.

[68] Eliseu M. Chaves Filho and Edil S. Tavares Fernandes. The effect of the speculation depth

on the performance of superscalar architectures. In *International Euro-Par Conference on Parallel Processing*, pages 1061–1065, 1997.

[69] Pengfei Gao, Jun Zhang, Fu Song, and Chao Wang. Verifying and quantifying side-channel resistance of masked software implementation. *ACM Trans. Softw. Eng. Methodol.*, 2019.

[70] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptographic Engineering*, 8 (1):1–27, 2018.

[71] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time protection: The missing OS abstraction. In *Proceedings of the Fourteenth EuroSys Conference*, pages 1:1–1:17, 2019.

[72] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[73] Philipp Grabher, Johann Großschädl, and Dan Page. Cryptographic side-channels from low-power cache memory. In *International Conference on Cryptography and Coding*, pages 170–184, 2007.

[74] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games–bringing access-based cache attacks on aes to practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505. IEEE, 2011.

[75] Shengjian Guo, Meng Wu, and Chao Wang. Adversarial symbolic execution for detecting concurrency-related cache timing leaks. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 377–388, 2018.

[76] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET

benchmarks – past, present and future. In *International Workshop on Worst-Case Execution Time Analysis*, pages 137–147, 2010.

[77] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE International Workshop on Workload Characterization*, pages 3–14, 2001.

[78] Daniel Hedin and David Sands. Timing aware information flow security for a javacard-like bytecode. *Electr. Notes Theor. Comput. Sci.*, 141(1):163–182, 2005.

[79] Wei-Ming Hu. Reducing timing channels with fuzzy time. In *IEEE Symposium on Security and Privacy*, pages 8–20, 1991.

[80] Bach Khoa Huynh, Lei Ju, and Abhik Roychoudhury. Scope-aware data cache analysis for WCET estimation. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2011, Chicago, Illinois, USA, 11-14 April 2011*, pages 203–212, 2011.

[81] Zhen Hang Jiang, Yunsi Fei, and David R. Kaeli. A complete key recovery timing attack on a GPU. In *IEEE International Symposium on High Performance Computer Architecture*, pages 394–405, 2016.

[82] Daniel A Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *IEEE International Symposium On High Performance Computer Architecture*, pages 197–206, 2001.

[83] Xiaoqing Jin, Jyotirmoy V. Deshmukh, James Kapinski, Koichi Ueda, and Ken Butts. Powertrain control verification benchmark. In *17th International Conference on Hybrid Systems: Computation and Control*, 2014.

[84] Xiaoqing Jin, Alexandre Donzé, Jyotirmoy V Deshmukh, and Sanjit A Seshia. Mining

requirements from closed-loop control models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(11):1704–1717, 2015.

[85] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, January 2018.

[86] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[87] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *International Cryptology Conference*, pages 388–397, 1999.

[88] Bettina Könighofer, Mohammed Alshiekh, Roderick Bloem, Laura R. Humphrey, Robert Könighofer, Ufuk Topcu, and Chao Wang. Shield synthesis. *Formal Methods in System Design*, 51(2):332–361, 2017.

[89] Boris Köpf and Markus Dürmuth. A provably secure and efficient countermeasure against timing attacks. In *IEEE Computer Security Foundations Symposium*, pages 324–335, 2009.

[90] Boris Köpf and Heiko Mantel. Transformational typing and unification for automatically correcting insecure programs. *Int. J. Inf. Sec.*, 6(2-3):107–131, 2007.

[91] Boris Köpf and Geoffrey Smith. Vulnerability bounds and leakage resilience of blinded cryptography under timing attacks. In *IEEE Computer Security Foundations Symposium*, pages 44–56, 2010.

[92] Markus Kusano and Chao Wang. Flow-sensitive composition of thread-modular abstract interpretation. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2016.

[93] Markus Kusano and Chao Wang. Thread-modular static analysis for relaxed memory models. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 337–348, 2017.

[94] Xianfeng Li, Tulika Mitra, and Abhik Roychoudhury. Accurate timing analysis by modeling caches, speculation and their interaction. In *ACM/IEEE Design Automation Conference*, pages 466–471, 2003.

[95] Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. Modeling out-of-order processors for wcet analysis. *Real-Time Systems*, 34(3):195–227, 2006.

[96] Yan Li, Vivy Suhendra, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *IEEE Real-Time Systems Symposium*, pages 57–67, 2009.

[97] J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM Trans. Inf. Syst. Secur.*, 12(3), 2009.

[98] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *ArXiv e-prints*, January 2018.

[99] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. *ACM SIGARCH Computer Architecture News*, 43(1):87–101, 2015.

[100] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 406–418. IEEE, 2016.

[101] Jun Liu, Necmiye Ozay, Ufuk Topcu, and Richard M Murray. Synthesis of switching protocols from temporal logic specifications. 2011.

[102] Fan Long, Stelios Sidiroglou-Douskos, and Martin C. Rinard. Automatic runtime error repair and containment via recovery shepherding. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 227–238, 2014.

[103] Q. Luo and G. Roşu. Enforcemop: a runtime property enforcement system for multithreaded programs. In *International Symposium on Software Testing and Analysis*, pages 156–166, 2013.

[104] Mingsong Lv, Wang Yi, Nan Guan, and Ge Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In *IEEE Real-Time Systems Symposium*, pages 339–349, 2010.

[105] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In *Joint International Conferences on Formal Modelling and Analysis of Timed Systems*, pages 152–166, 2004.

[106] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks - Revealing the Secrets of Smart Cards*. Springer, 2007. ISBN 978-0-387-30857-9.

[107] Heiko Mantel and Artem Starostin. Transforming out timing leaks, more or less. In *European Symposium on Research in Computer Security*, pages 447–467, 2015.

[108] Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In *European Symposium on Programming Languages and Systems*, pages 5–20, 2005.

[109] R. Mazala. Infinite games. In *Automata, Logics, and Infinite Games: A Guide to Current Research*, LNCS 2500, 2001.

[110] Ravindra Metta, Martin Becker, Prasad Bokil, Samarjit Chakraborty, and R Venkatesh. Tic: a scalable model checking based approach to wcet estimation. In *ACM SIGPLAN Notices*, volume 51, pages 72–81. ACM, 2016.

[111] Jonathan K. Millen. Covert channel capacity. In *IEEE Symposium on Security and Privacy*, pages 60–66, 1987.

[112] Antoine Miné. The octagon abstract domain. *Higher-order and symbolic computation*, 19 (1):31–100, 2006.

[113] Tulika Mitra, Jürgen Teich, and Lothar Thiele. Time-critical systems design: A survey. *IEEE Design & Test*, 35(2):8–26, 2018.

[114] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *International Conference on Information Security and Cryptology*, pages 156–168. Springer, 2005.

[115] Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. Compiler assisted masking. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 58–75, 2012.

[116] Keaton Mowery, Sriram Keelveedhi, and Hovav Shacham. Are aes x86 cache timing attacks still feasible? In *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop*, pages 19–24. ACM, 2012.

[117] NHTSA. *49 CFR Part 571: Federal Motor Vehicle Safety Standards; Accelerator Control Systems*. Department of Transportation, 2012.

[118] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 2010. ISBN 3642084745, 9783642084744.

[119] Shirin Nilizadeh, Yannic Noller, and Corina S. Pasareanu. DifFuzz: differential fuzzing for side-channel analysis. In *International Conference on Software Engineering*, pages 176–187, 2019.

[120] Petter Nilsson, Omar Hussien, Ayca Balkan, Yuxiao Chen, Aaron D Ames, Jessy W Grizzle, Necmiye Ozay, Huei Peng, and Paulo Tabuada. Correct-by-construction adaptive cruise control: Two approaches. *IEEE Transactions on Control Systems Technology*, 24(4):1294–1307, 2016.

[121] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, pages 1–20, 2006. doi: 10.1007/11605805_1. URL https://doi.org/10.1007/11605805_1.

[122] D. Page. Partitioned cache architecture as a side-channel defence mechanism, 2005. URL http://eprint.iacr.org/2005/280. page@cs.bris.ac.uk 13017 received 22 Aug 2005.

[123] Yash Vardhan Pant, Houssam Abbas, Rhudii A Quaye, and Rahul Mangharam. Fly-by-logic: control of multi-drone fleets with temporal logic objectives. In *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*, pages 186–197. IEEE Press, 2018.

[124] Corina S. Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. Multi-run side-channel analysis using symbolic execution and max-smt. In *IEEE Computer Security Foundations Symposium*, pages 387–400, 2016.

[125] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis for C. In *Proceedings of the 2004 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'04, Washington, DC, USA, June 7-8, 2004*, pages 37–42, 2004. doi: 10.1145/996821.996835. URL http://doi.acm.org/10.1145/996821.996835.

[126] Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, and Tevfik Bultan. Synthesis of adaptive side-channel attacks. In *IEEE Computer Security Foundations Symposium*, pages 328–342, 2017.

[127] Jim Pierce and Trevor N. Mudge. The effect of speculative execution on cache performance. In *International Symposium on Parallel Processing*, pages 172–179, 1994.

[128] Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. Probabilistic timing covert channels: to close or not to close? *Int. J. Inf. Sec.*, 10(2):83–106, 2011.

[129] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 179–190, 1989.

[130] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, LNCS 137. Springer, 1982.

[131] Vasumathi Raman, Alexandre Donzé, Mehdi Maasoumy, Richard M Murray, Alberto Sangiovanni-Vincentelli, and Sanjit A Seshia. Model predictive control with signal temporal logic specifications. In *Decision and Control (CDC), 2014 IEEE 53rd Annual Conference on*, pages 81–87. IEEE, 2014.

[132] Vasumathi Raman, Alexandre Donzé, Dorsa Sadigh, Richard M Murray, and Sanjit A Seshia. Reactive synthesis from signal temporal logic specifications. In *Proceedings of the 18th*

international conference on hybrid systems: Computation and control, pages 239–248. ACM, 2015.

[133] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 431–446, 2015.

[134] Matthieu Renard, Yliès Falcone, Antoine Rollet, Srinivas Pinisetty, Thierry Jéron, and Hervé Marchand. Enforcement of (timed) properties with uncontrollable events. In *International Colloquium on Theoretical Aspects of Computing*, pages 542–560, 2015.

[135] Matthieu Renard, Antoine Rollet, and Yliès Falcone. Runtime enforcement using büchi games. In *ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, pages 70–79, 2017.

[136] Matthieu Renard, Yliès Falcone, Antoine Rollet, Thierry Jéron, and Hervé Marchand. Optimal enforcement of (timed) properties with uncontrollable events. *Mathematical Structures in Computer Science*, 29(1):169–214, 2019.

[137] Martin C. Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 303–316, 2004.

[138] Nima Roohi, Ramneet Kaur, James Weimer, Oleg Sokolsky, and Insup Lee. Parameter invariant monitoring for signal temporal logic. In *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (part of CPS Week)*, pages 187–196. ACM, 2018.

[139] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3:30–50, 2000.

[140] Jörn Schneider. Statische pipeline-analyse für echtzeitsysteme. *Dipl. thesis, Univ. Saarland, Saarbruecken, Germany*, 1998.

[141] Jörn Schneider and Christian Ferdinand. Pipeline behavior prediction for superscalar processors by abstract interpretation. In *ACM SIGPLAN Notices*, volume 34, pages 35–44, 1999.

[142] Bruce Schneier. *Applied cryptography: protocols, algorithms, and source code in C*. john wiley & sons, 2007.

[143] Rathijit Sen and Y. N. Srikant. WCET estimation for executables in the presence of data caches. In *ACM/IEEE International conference on Embedded Software*, pages 203–212, 2007.

[144] Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 1948.

[145] Geoffrey Smith. On the foundations of quantitative information flow. In *International Conference on the Foundations of Software Science and Computational Structures*, pages 288–302, 2009.

[146] Saqib Sohail and Fabio Somenzi. Safety first: a two-stage algorithm for the synthesis of reactive systems. *International Journal on Software Tools for Technology Transfer*, 15(5-6): 433–454, 2013.

[147] Marcelo Sousa and Isil Dillig. Cartesian hoare logic for verifying k-safety properties. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 57–69, 2016.

[148] Raphael Spreitzer and Thomas Plos. On the applicability of time-driven cache attacks on

mobile devices. In *International Conference on Network and System Security*, pages 656–662. Springer, 2013.

[149] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *ACM SIGSAC Conference on Computer & Communications Security*, pages 299–310, 2013.

[150] Chungha Sung, Markus Kusano, and Chao Wang. Modular verification of interrupt-driven software. In *IEEE/ACM International Conference On Automated Software Engineering*, pages 206–216, 2017.

[151] Chungha Sung, Brandon Paulsen, and Chao Wang. CANAL: a cache timing analysis framework via LLVM transformation. In *IEEE/ACM International Conference On Automated Software Engineering*, pages 904–907, 2018.

[152] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. Fast and precise wcet prediction by separated cache and path analyses. *Real-Time Systems*, 18(2):157–179, 2000.

[153] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, Jan 1967.

[154] Valentin Touzeau, Claire Maiza, David Monniaux, and Jan Reineke. Ascertaining uncertainty for efficient exact cache analysis. In *International Conference on Computer Aided Verification*, pages 22–40, 2017.

[155] Cumhur Erkan Tuncali, James Kapinski, Hisahiro Ito, and Jyotirmoy V. Deshmukh. Reasoning about safety of learning-enabled components in autonomous cyber-physical systems. In *ACM/IEEE Design Automation Conference*, pages 30:1–30:6, 2018.

[156] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow:

Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security Symposium*, 2018.

[157] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in xen. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 41–46. ACM, 2011.

[158] Lucian N Vintan and Mihaela Iridon. Towards a high performance neural branch predictor. In *International Joint Conference on Neural Networks*, pages 868–873, 1999.

[159] Chao Wang and Patrick Schaumont. Security by compilation: an automated approach to comprehensive side-channel resistance. *SIGLOG News*, 4(2):76–89, 2017.

[160] Jingbo Wang, Chungha Sung, and Chao Wang. Mitigating power side channels during compilation. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[161] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 494–505, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-706-3. doi: 10.1145/1250662.1250723. URL http://doi.acm.org/10.1145/1250662.1250723.

[162] Reinhard Wilhelm, Sebastian Altmeyer, Claire Burguière, Daniel Grund, Jörg Herter, Jan Reineke, Björn Wachter, and Stephan Wilhelm. Static timing analysis for hard real-time systems. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 3–22, 2010.

[163] Meng Wu, Haibo Zeng, and Chao Wang. Synthesizing runtime enforcer of safety properties

under burst error. In *NASA Formal Methods - 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings*, pages 65–81, 2016.

[164] Meng Wu, Haibo Zeng, Chao Wang, and Huafeng Yu. Safety guard: Runtime enforcement for safety-critical cyber-physical systems: Invited. In *ACM/IEEE Design Automation Conference*, pages 84:1–84:6, 2017.

[165] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. Eliminating timing side-channel leaks using program repair. In *International Symposium on Software Testing and Analysis*, pages 15–26, 2018.

[166] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 719–732, 2014. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

[167] Tse-Yu Yeh and Yale N Patt. Two-level adaptive training branch prediction. In *IEEE/ACM International Symposium on Microarchitecture*, pages 51–61, 1991.

[168] F. Yu, M. Alkhalaf, and T. Bultan. Patching vulnerabilities with sanitization synthesis. In *International Conference on Software Engineering*, pages 251–260, 2011.

[169] Danfeng Zhang, Aslan Askarov, and Andrew C Myers. Language-based control and mitigation of timing channels. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 99–110, 2012.

[170] Jun Zhang, Pengfei Gao, Fu Song, and Chao Wang. SCInfer: Refinement-based verification of software countermeasures against side-channel attacks. In *International Conference on Computer Aided Verification*, pages 157–177, 2018.

[171] Lu Zhang and Chao Wang. Runtime prevention of concurrency related type-state violations in multithreaded applications. In *International Symposium on Software Testing and Analysis*, pages 1–12, 2014. doi: 10.1145/2610384.2610405. URL http://doi.acm.org/10.1145/2610384.2610405.

[172] Naling Zhang, Markus Kusano, and Chao Wang. Dynamic partial order reduction for relaxed memory models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 250–259, 2015.