SCF^{MSP}: Static Detection of Side Channels in MSP430 Programs

Sepideh Pouyanrad
imec-DistriNet, KU Leuven
sepideh.pouyanrad@cs.kuleuven.be

Jan Tobias Mühlberg
imec-DistriNet, KU Leuven
jantobias.muehlberg@cs.kuleuven.be

Wouter Joosen
imec-DistriNet, KU Leuven
wouter.joosen@cs.kuleuven.be

ABSTRACT

Information leakage through side-channels poses a serious threat to the security of distributed systems. Recent research on countermeasures against side-channel attacks show that, on embedded platforms with predictable execution times, certain classes of these vulnerabilities can be detected and mitigated automatically by means of language-based security techniques. In this paper, we propose a security type system to statically analyse MSP430 assembly programs to detecting information leakage through novel interrupt-latency attacks (a.k.a. Nemesis), timing side-channels, and undesired information flow. We have implemented our technique in a tool, *Side Channel Finder^{MSP}*, which automatically verifies MSP430 object-code programs to be free of such vulnerabilities. We evaluate the effectiveness of our tool by applying it to a representative set of vulnerable and benign programs. Our experiments demonstrate that the tool is both effective in detecting vulnerabilities, and scalable to realistic applications.

CCS CONCEPTS

• **Security and privacy → Security in hardware**; Embedded systems security; *Formal methods and theory of security*; Software and application security;

KEYWORDS

Side channel analysis, Embedded devices, Language-Based security, MSP430, Assembly programs, Nemesis attack, Static analysis

ACM Reference Format:
Sepideh Pouyanrad, Jan Tobias Mühlberg, and Wouter Joosen. 2020. SCF^{MSP}: Static Detection of Side Channels in MSP430 Programs. In *The 15th International Conference on Availability, Reliability and Security (ARES 2020), August 25–28, 2020, Virtual Event, Ireland*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3407023.3407050

1 INTRODUCTION

The increased pervasiveness of embedded computing systems in a wide range of application environments have made these systems attractive targets for attackers [1, 5, 39]. Attacks on embedded systems can cause physical damages to infrastructures, lead to the extraction and dispersion of confidential information, cause to system malfunction, disrupt digital transactions and involve financial loss, and allow for unauthorised access to resources. Over the past decade, side channel attacks have become an acknowledged cyber threat to many embedded computing systems. Side-channel attacks aim at extracting sensitive information from a system through measurement and analysis of its execution characteristics like timing information, cache behaviour, power consumption, electromagnetic radiation or other physical side effects [18, 25, 26, 30, 35].

In this paper, we present a security type system to statically verify assembly programs for the TI MSP430 microcontroller for the absence of timing side channel attacks, interrupt-latency attacks (i.e., Nemesis [14]), and undesired direct and indirect information flow. We implemented our proposed security type system in an automatic tool SCF^{MSP}, named after [16]. To demonstrate the effectiveness of our approach, we evaluate SCF^{MSP} in a series of experiments on example programs that exhibit typical and challenging structures of secret-dependent control flow. We show that SCF^{MSP} can identify vulnerabilities in these programs, and also verify the absence of information leakage in manually repaired versions of the examples. Our goal in this paper is to help programmers develop programs free of side channel by automatically analysing that the resource usage variations of a given program does not vary with respect to the secret.

The TI MSP430 microcontroller is widely used for low-powered embedded devices that has a number of special features not commonly available in other microcontrollers. Over the past few years, researchers have proposed different security architectures for microcontrollers. Sancus [33, 34], a hardware-only Trusted Execution Environment (TEE), implements memory protection and extensions to the instruction set of the MSP430 to provide strong security guarantees with a hardware-only Trusted Computing Base (TCB). While TEEs substantially improve software security in embedded systems, side-channels are typically outside of the attacker model of these architectures and pose a demonstrable threat to applications.

The MSP430 serial bootstrap loader (BSL) v2.12 has been attacked successfully through a timing side channel attack [20]. Recently, researchers have introduced Nemesis [14], a powerful microarchitectural side-channel attack that reveals instruction timings from a Sancus enclave by exploiting the CPU's interrupt mechanism. Additionally, many considerations of side channel attacks on the MSP430FR5969 have been presented in [32]. The authors have shown that a couple of different power analysis attacks against the hardware AES peripheral, as well as the trace-driven cache attacks are expectedly feasible. Arguable, however, software-based channels such as Nemesis are more dangerous because the attack can be implemented entirely in software, without physical access to a system. Effective countermeasures against side channel attacks are well known in theory, but in practice are challenging to implement properly. Hence, side channel vulnerabilities continue to be uncovered in the real-world security-critical embedded systems.

ARES 2020, August 25–28, 2020, Virtual Event, Ireland

ACM ISBN 978-1-4503-8833-7/20/08…$15.00
https://doi.org/10.1145/3407023.3407050

In this paper, we focus on detecting, rather than mitigating, side channel attacks based on a security type system. This approach has previously been proposed for multiple sequential and concurrent programming languages [3, 9, 27, 40]. Advanced architectural features such as paging, caches, branch prediction, or out-of-order instruction pipelining is not considered in the theoretical evaluation of these techniques. More specifically, these type systems are not completely effective for removing side channel attacks in modern high-end processors [31].

Contributions. Prior work by Mantel et al. [16] provides the first security type system to verify AVR (a popular microcontroller) assembly programs for the absence of timing side channel vulnerabilities in a sound way. Our work builds upon [16] to support a different popular microcontroller and a novel type of side channel. Specifically, we make the following contributions:

- We develop a formal operational semantics for MSP430 instructions set based on [22] that reflect execution time and information flow.
- We propose a security type system to statically detect timing side channel attacks, Nemesis [14] attacks and undesired information flow in MSP430 assembly programs.
- We implement the proposed method in a software tool SCF^{MSP}, written in Python, to automatically check program binaries and report their side channel vulnerabilities. We make a repository with SCF^{MSP} and our benchmarks publicly available at https://github.com/sepidehpouyan/SCF-MSP430.
- We evaluated the accuracy and the scalability of the tool by applying to a representative set of vulnerable and benign programs.

Organization. The paper is structured as follows. We start by giving an overview of side channel attacks and explain our threat model in Section 2, followed by a discussion or related work on the detection of side-channel information leakage in programs in Section 3. In Section 4, we elaborate on our semantics of MSP430 assembler, on top of which we develop security properties and a security type system. We then describe the components of our SCF^{MSP}, and their tasks in Section 5. In Section 6, we present the results of our evaluation. Finally, we conclude our work and outline future directions of research.

2 PROBLEM STATEMENT

In this section, we give a brief description of timing side channels that we focus on this paper with the aid of a real-world example and explain the threat model that we assume throughout the paper. Finally, we briefly review features and functions of MSP430 architecture.

2.1 Timing Side Channel Attacks

An illustrative example of a timing side channel can be found in the password comparison routine of TI MSP430 serial Bootstrap Loader (BSL) implementation [20]. The BSL software allows users to program the embedded memory in the MSP430 microcontroller during project development and updates. For security purposes and to prevent unwanted source readout, sensitive BSL commands that directly or indirectly allow data reading is protected by a user-defined 32-byte password.

Listing 1: Unbalanced (i.e., vulnerable) BSL password comparison from [14].

```
1 cmp.b @r6+, r12
2 jz    1f
3 bis   #0x40, r11
4 1: ...
5
```

Listing 2: Balanced version of Listing 1, from [14].

```
1 cmp.b @r6+, r12
2 jz    1f
3 bis   #0x40, r11
4 jmp   2f
5 1: nop nop nop nop
6 2: ...
```

Listing 1 shows a fragment of the BSL code, where the value pointed to by `r6` is compared with `r12`. A bit in `r11` is set when the comparison fails. Observe that the code in Listing 1 is unbalanced since processing an incorrect password byte takes two clock cycles (the `bis` ("bit-set") instruction, line 3) longer than a correct byte. Hence, the program's overall execution time varies depending on the correctness of individual bytes [20]. In order to equalise the timing behaviour of both branches, dummy instructions (Listing 2, line 5) are appended in a balanced version of the code, following the methodology of [3, 15]. Thus, the timing leak is properly closed. However, the authors in [14] have shown that even after removing the dependencies between execution time and password, attackers can still determine the correctness of individual password bytes by a previously overlooked side channel attack called Nemesis [14].

2.2 Nemesis-type Interrupt Timing Attacks

Nemesis attacks abuse the CPU's interrupt mechanism to gain side-channel information about the interrupted instruction [14]. In most multi-cycle instruction set architectures, interrupt requests are only served after completing the current executing instruction. This interrupt handling mechanism introduces a timing difference between the interrupted instructions that can be exploited by an attacker to differentiate between secret-dependent branches. Nemesis is particularly relevant in the context of embedded processors with Trusted Execution features such as [34], where the attack is capable of extracting application secrets from vulnerable enclaves.

The BSL password comparison routine, Figure 2, is vulnerable to Nemesis-type attacks. Specifically, an interrupt request arriving in the first clock cycle after the conditional jump instruction, will either interrupt the two-cycle `bis` instruction or the single-cycle `nop` instruction. Here, an attacker can observe an interrupt latency difference of one clock cycle depending on whether the `jz` instruction was taken or not and finally determine the value of password. Moreover, the number of times that the if/else branch can be interrupted will be measured by attackers [14]. Concretely, for launching a successful Nemesis attack on processors with constant-time interrupt latency and multi-cycle instruction set, an attacker just requires at least one instruction with a different execution time or different count of instructions in the if/else branch.

2.3 Attacker Model

In side channel attacks, the attackers attempt to collect information regarding resource usages of critical-safety embedded systems. An attacker model provides a precise definition of the power and capability of the attacker. In this paper, we consider an attacker

with access to the source code or the compiled binary code of the victim's program. We further assume that attackers can observe program's execution time and outputs. We also consider a more capable attacker with two abilities to configure hardware devices such as timers and to deploy malicious modules to embedded devices. Physical side channels, including power consumption and electromagnetic radiation, are out of scope and are covert in orthogonal research. Side channels caused by the microarchitecture such as cache contention and branch prediction are out of the scope of this work because these features are not present in the MSP430 target architecture.

A realistic scenario with the aforementioned properties could be a microcontroller that the attacker and the victim connect to through a network. The victim engages with the microcontroller to run a critical application. By observing the network and measuring the critical application's start-to-end execution time, communication patterns, or by deploying a spy module and dynamically configuring timers and interrupts to launch a Nemesis attack [14], the attacker would be capable of inferring the state of the victim program.

2.4 TI MSP430 Architecture

The TI MSP430 is a 16-bit microcontroller from Texas Instruments that is designed for low power consumption embedded devices. An MSP430 CPU [22] incorporates sixteen 16-bit registers including program counter, status register, stack pointer, constant generator and 12 registers for general purpose. The constant generator provides six most commonly-used constants for the assembler and reduces code size, required memory access to retrieve the immediate value and consequently the instruction's execution time. Furthermore, the CPU features seven addressing modes for the source operand and four addressing modes for the destination operand with a complete instruction set consisting of 27 core instructions and 24 emulated instructions. The emulated instructions do not have op-codes. They are replaced automatically by the assembler with an equivalent core instruction.

The time required to execute an instruction on an MSP430 microcontroller, in the absence of advanced architectural features such as paging, caches, or out-of-order instruction pipelining, is completely deterministic. More specifically, instruction executes between one and six clock cycles, depending on the addressing modes of the source and destination operands. The soundness of our proposed security type system relies on this feature which makes an instruction's execution time accurately predictable. We refer to [22] for a comprehensive description of MSP430 specifications.

3 RELATED WORK

Timing Side Channels. Kocher [26] is probably the first to publicly introduce microarchitectural timing side channels against standard cryptosystems such as RSA, Diffie-Hellman, and DSS. Since then, there exists a vast body of work on the timing side channel attacks and countermeasures in different platforms [19]. For example, Brumley et al. [12] conducted practical remote timing attacks through a computer network. Bernstein [10] demonstrated the feasibility of cache collision timing attacks against AES executing on modern processors. Over the past two decades, it has furthermore been shown that an attacker can extract confidential data by exploiting the cache contention using several practical attack techniques, including Evict+Time, Prime+Probe [36], and Flush+Reload [42].

Aciiçmez et al. [2] presented a new microarchitectural side channel attack by abusing CPU's branch prediction machinery to attack the modular inversion during RSA computations. The prominent Spectre [24] and Meltdown [29] attacks exploit out of order execution and speculative execution resulting from a branch misprediction, respectively, to execute transient instructions and consequently gain unauthorised access to confidential data. These two vulnerabilities are considered "catastrophic" by security analysts because of disruptive impact on most processors. Timing side channels caused by advanced microarchitectural CPU features such as paging, caching, branch prediction, or out-of-order execution are not considered in this paper because these features are not supported in MSP430 architecture.

Timing Side Channels on MSP430. In 2008, Travis [20] showed the secret-dependence branch in the password comparison routine of MSP430 BSL v2.12 is vulnerable to an execution timing attack. Recently, Jo et al. [14] have presented Nemesis, an innovative timing side channel attack enabled by the key microarchitectural property that hardware interrupts are delayed until currently executing instruction retirement. This microarchitectural behaviour introduces a timing difference when interrupting different instructions. They further carried out a successful Nemesis interrupt latency attack against the Sancus [33] embedded processor to differentiate between secret-dependent program branches by timing measurement.

Information Flow Analysis. There has been a significant body of work on type-based solutions for enforcing timing-sensitive information flow properties and we will only consider those approaches closely related to our work. Volpano and Smith [40] proposed a security type system in which well-type programs are non-interfering and guarantees absence of timing leakage fundamentally. Though effective, this approach is too restrictive and thus all the existing programs need to be rewritten to follow its specification. More specifically, it forbids the sequential programs with looping and branching conditions dependent on secret data. Agat [3] provided a type system to eliminate timing side channels via program transformation. Basically, by inserting padding `nop` instructions in places identified by the type system, he can ensure that secret-dependence branches have the same timing characteristics, hence removing all timing leaks. Barthe et al. [9] extended Agat's work towards a sequential language with objects and exceptions by developing a transaction-based program transformation technique that leverages commit/abort operations. Independently, Köpf and Mantel [27] proposed a type system to remove timing leaks by incorporating unification. However, Mantel et al. [31] compared four of the existing transformation techniques on Java byte-code, and demonstrated that none was able to remove the leaks completely.

Information Flow Control techniques can be applied down to the hardware to address timing side channels. For instance, Zhang et al. [43] designed a hardware design language, SecVerilog, with a security type system which uses static information flow tracking to verify side channel leakage at hardware level. Sapper [28], instead

of static verification, adds information flow tracking logic into the original hardware.

Leakage at assembly-level is also considered by Barthe et al. [6]. They suggested a static information-flow analysis to verify the constant-time policy in the C programs were compiled by CompCert to an abstraction of assembly. In 2015, Doychev et al. [17] developed an automatic tool named CacheAudit that is based on the static binary analysis for detection of cache timing side channels in x86 binaries. Wichelmann et al. [41] develop a framework named MicroWalk to measure the correlation between critical secrets and runtime behaviours from a subset of the execution traces based on the dynamic program analysis techniques, capturing side channels in binaries.

The work most closely related to ours is presented by Mantel et al. [16] They proposed the first information flow analysis and automatic analysis tool to statically verify that AVR assembly programs are free of timing side channels. Based on the predictability of instruction's execution times on 8-bit AVR processors, they proved their type system's soundness. Furthermore, they showed their tool is applicable to check information leakage in the real-world crypto library μNaCl. In a similar way, we provide a security type system to guarantee the absence of timing leaks and as well as Nemesis-style vulnerability in MSP430 assembly programs. In contrast to AVR microprocessor, we must observe different addressing modes and CGs (Constant Generators) in the formal semantics to include timing information for each MSP430 instruction. Moreover, we developed the first static analysis tool to automatically analyse MSP430 assembly programs for detecting timing side channels and Nemesis.

4 A SEMANTICS AND SECURITY TYPE SYSTEM FOR MSP430 PROGRAMS

We define a formal semantics of MSP430 assembly programs that we then use as basis for illustrating the construction of our security properties and for proposing a security type system. More concretely, we provide a mathematical model of the MSP430 assembly program that describes its execution and procedures. To the best of our knowledge, we present the first formal semantics for MSP430 assembly programs.

4.1 Syntax

Before setting down our definitions for the terminologies and notations that we shall consider, we first define the syntax of MSP430 assembly instructions that we shall apply in our security type system, with the standard grammar BNF given in Figure 1. Each MSP430 instruction can have one of the following three formats [22]: *(1)* instructions requiring two operands, *(2)* instructions requiring a single operand, and *(3)* instructions whose operands are relative offsets. However, we categorise instructions based on the number and type of arguments in our syntax. To account for the limitations of our static analysis, we only accept jump instructions with constant target locations. The remaining instructions can operate on registers, memory, and immediate values, depending on the addressing mode.

INSTR	::=	Single src | Double src dst | Jump val | Return
src	::=	Rs | X(Rs) | val | & val | @ Rs | @ Rs + | # val
dst	::=	Rd | X(Rd) | val | & val
Rd, Rs	::=	Rn | n ∈ [0, 15]
val	::=	$[0, 2^{16}]$
X	::=	val
addr _bit	::=	As | Ad | ∅
Ad	::=	0 | 1
As	::=	[v1 , v2]
v	::=	0 | 1
Where		
Single	∈	{ RRC, SWPB, RRA, SXT, PUSH },
Double	∈	{MOV, ADD, ADDC, SUBC, SUB, CMP, BIT, BIS, BIC, XOR, AND, BIT, BIS, BIC, XOR, AND},
Jump	∈	{ JNZ, JZ, JNC, JC, JN, JGE, JL, JMP, CALL},
Return	∈	{RETI}.

Figure 1: The syntax.

4.2 Semantics

We capture the individual execution steps of a MSP430 assembly program, and describe states and execution time through a small-step operational semantics defined for each instruction. In order to obtain the source and destination addressing modes in MSP43o, we define a function $MODE : addr\_bit \rightarrow [0, 7]$ where *addr_bit* model the addressing bits responsible for the addressing mode in an instruction binary format. Let ∅ model the empty input. We further employ a function $CG : (src, As) \rightarrow 0, 1$ to model the effect of MSP430 constant generator registers (*R2* and *R3*) in reducing execution time and length of instructions.

We obtain the required number of clock cycles that each instruction takes by a function $t : (INSTR, MODE, MODE, CG) \rightarrow N$ and use a function $ep_{\text{next}} : (INSTR, MODE, MODE, CG) \rightarrow val$ to calculate next execution point based on the length of instructions. We define these functions based on Table 1 and Table 2 that list the number of clock cycles and the length for respectively single and double operand instructions of the MSP430 architecture. All jump instructions require one code word and take two clock cycles to execute, regardless of whether the jump is taken or not. The length and the number of CPU cycles required for other instructions depends on the addressing modes of the source and destination operands, not the instruction type itself.

Following prior work in [16], the set **State** of possible program's execution states is defined as the set of the forms (*sr, m, st, r, st, ep*) where *sr* is a set that models the contents of the dedicated state register that stores status flags (*C, Z, N, O*), and *m* is a meta variable for the memory contents, and *r* and *st* represent the contents of the registers and the stack respectively. We let *ep* denote the program's execution point, specifically, the address of current executing instruction that is kept in the program counter (*PC/R0*). Let a program P(ep) := INSTR be a mapping from execution points to instructions. Finally, we model transitions between consecutive states through a relation $s \xrightarrow{t}_P \acute{s}$, where $s, \acute{s} \in$ **State**, to declare that s leads to $\acute{s}$ in time *t*. The evaluation judgements defined for instructions PUSH,

Table 1: MSP430 single operand instruction according to [22].

Addressing Mode	No. of Cycles: RRA, RRC SWPB, SXT	No. of Cycles: PUSH	No. of Cycles: CALL	Length of Instruction	Example
R_n	1	3	4	1	`SWPB R5`
@R_n	3	4	4	1	`RRC @R9`
@R_n+	3	5	5	1	`SWPB @R10+`
#N	-	4	5	2	`CALL #0F000h`
X(R_n)	4	5	5	2	`CALL 2(R7)`
EDE	4	5	5	2	`PUSH EDE`
&EDE	4	5	5	2	`SXT &EDE`

Table 2: MSP430 double operand instruction [22].

Addressing Mode Src	Addressing Mode Dst	No. of Cycles	Length of Instruction	Example
R_n	R_m	1	1	`MOV R5,R8`
	PC	2	1	`BR R9`
	x(R_m)	4	2	`ADD R5,4(R6)`
	EDE	4	2	`XOR R8,EDE`
	&EDE	4	2	`MOV R5,&EDE`
@R_n	R_m	2	1	`AND @R4,R5`
	PC	2	1	`BR @R8`
	x(R_m)	5	2	`XOR @R5,8(R6)`
	EDE	5	2	`MOV @R5,EDE`
	&EDE	5	2	`XOR @R5,&EDE`
@R_n+	R_m	2	1	`ADD @R5+,R6`
	PC	3	1	`BR @R9+`
	x(R_m)	5	2	`XOR @R5,8(R6)`
	EDE	5	2	`MOV @R9+,EDE`
	&EDE	5	2	`MOV @R9+,&EDE`
#N	R_m	2	2	`MOV #20,R9`
	PC	3	2	`BR #2AEh`
	x(R_m)	5	3	`MOV #0300h,0(SP)`
	EDE	5	3	`ADD #33,EDE`
	&EDE	5	3	`ADD #33,&EDE`
x(R_n)	R_m	3	2	`MOV 2(R5),R7`
	PC	3	2	`BR 2(R6)`
	TONI	6	3	`MOV 4(R7),TONI`
	x(R_m)	6	3	`ADD 4(R4),6(R9)`
	&TONI	6	3	`MOV 2(R4),&TONI`
EDE	R_m	3	2	`AND EDE,R6`
	PC	3	2	`BR EDE`
	TONI	6	3	`CMP EDE,TONI`
	x(R_m)	6	3	`MOV EDE,0(SP)`
	&TONI	6	3	`MOV EDE,&TONI`
&EDE	R_m	3	2	`MOV &EDE,R8`
	PC	3	2	`BR &EDE`
	TONI	6	3	`MOV &EDE,TONI`
	x(R_m)	6	3	`MOV &EDE,0(SP)`
	&TONI	6	3	`MOV &EDE,&TONI`

$$\text{PUSH:}\ \frac{\begin{array}{c}P(ep) := PUSH\ \ Rs \\ st\text{fi} := r(Rd) \\ r\text{fi} := r(SP) - 1 \\ ep\text{fi} := ep_{\text{next}}(PUSH, MODE(As), MODE(\varnothing), CG)\end{array}}{(sr, m, r, st, ep) \xrightarrow{\text{t(PUSH, MODE(As), MODE(} \varnothing \text{), CG)}}_P (sr, m, r\text{fi}, st\text{fi}, ep\text{fi})}$$

$$\text{ADDC:}\ \frac{\begin{array}{c}P(ep) := ADDC\ \ Rs,\ \ x(Rd) \\ m\text{fi} := m(x(Rd)) + r(Rs) + sr(C) \\ sr\text{fi} := sr(C\text{fi}, Z\text{fi}, N\text{fi}, O\text{fi}) \\ Z\text{fi} := \neg(m\text{fi}(x(Rd))[15] \vee m\text{fi}(x(Rd))[14] \vee \ldots \vee (m\text{fi}(x(Rd))[0]) \\ C\text{fi} := (m(x(Rd))[15] \wedge r(Rs)[15]) \vee (r(Rs)[15] \wedge m\text{fi}(x(Rd))[15]) \vee next \\ next := (\neg m\text{fi}(x(Rd))[15] \wedge m(x(Rd))[15]) \\ N\text{fi} := m\text{fi}(x(Rd))[15] \\ O\text{fi} := C\text{fi}[14] \oplus C\text{fi}[15] \\ ep\text{fi} = ep_{\text{next}}(ADDC, MODE(As), MODE(Ad), CG)\end{array}}{(sr, m, r, st, ep) \xrightarrow{\text{t(ADDC, MODE(As), MODE(Ad), CG)}}_P (sr\text{fi}, m\text{fi}, r, st, ep\text{fi})}$$

$$\text{MOV:}\ \frac{\begin{array}{c}P(ep) := MOV\ \ @Rs,\ \ Rd \\ r\text{fi}(Rd) := m(@Rs) \\ ep\text{fi} := ep_{\text{next}}(MOV, MODE(As), MODE(Ad), CG)\end{array}}{(sr, m, r, st, ep) \xrightarrow{\text{t(MOV, MODE(As), MODE(Ad), CG)}}_P (sr, m, r\text{fi}, st, ep\text{fi})}$$

$$\text{JNZ-F:}\ \frac{\begin{array}{c}P(ep) := JNZ\ \ \ val \\ sr(Z) \neq 1 \\ ep\text{fi} := ep + 1\end{array}}{(sr, m, r, st, ep) \xrightarrow{\text{t(JNZ, MODE(} \varnothing \text{), MODE(} \varnothing \text{), CG)}}_P (sr, m, r, st, ep\text{fi})}$$

$$\text{JNZ-T:}\ \frac{\begin{array}{c}P(ep) := JNZ\ \ \ val \\ sr(Z) := 1 \\ ep\text{fi} := val\end{array}}{(sr, m, r, st, ep) \xrightarrow{\text{t(JNZ MODE(} \varnothing \text{), MODE(} \varnothing \text{), CG)}}_P (sr, m, r, st, ep\text{fi})}$$

Figure 2: Semantics of selected instructions.

ADDC, MOV, and JNZ of our small-step operational semantics are presented in Figure 2 as examples.

The MSP430 instruction "`PUSH Rs`" stores the content of the operand at the memory address pointed by the dedicated stack pointer register (R1 in MSP430). We define the resulting stack by st', which is updated to the content of *Rs*. The content of the stack pointer register is decremented by 1. We capture the execution time of PUSH by the annotation $t(PUSH, MODE(As), MODE(\varnothing), CG)$. Clearly speaking, in this rule, PUSH instruction operates on the content of the register (As := 00). Thereby, we capture the execution time by $t(PUSH, 1, 0, 0)$, which is 3. Since $ep_{next}(PUSH, 1, 0, 0) := 1$, we define the resulting execution point by ep' where $ep' := ep + 1$.

The MSP430 instruction "`ADDC Rs, x(Rd)`" stores the sum of the operands and the carry flag at the memory address pointed by $(r(Rd) + x)$. We capture the execution time of ADDC by the

annotation $t(ADDC, MODE(As), MODE(Ad), CG)$. This instruction depending on the source and destination addressing mode takes some deterministic times which is $t(ADDC, 1, 5, 0) = 4$ in this example, based on the Table 2. The execution point of the resulting state depends on the length of instruction ADDC that we capture by $ep_{next}(ADDC, MODE(As), MODE(Ad), CG)$. We define the resulting status flags by sr', which maps C to 1 if there was a carry, which maps Z to 1 if the sum is zero, which maps N to 1 if the sum is negative, and which maps O to 1 if overflow occurs. The resulting contents of $m(r(Rd) + x)$ is the sum of the contents of the *Rs*, pointed by *(r(Rd)+x)*, and *C*.

The MSP430 instruction "`JNZ val`" performs branching based on the zero flag. It takes 2 clock cycles and 1 code word, regardless of whether the zero flag is set or not [20]. The semantics of JNZ is defined by two rules based on the condition of the zero flag. All components of the state, except the execution point, are left unmodified in both rules. In the "`JNZ-F`" rule, since the jump is not taken, we capture the resulting execution point by $ep + 1$. However, the resulting execution point in the "`JNZ-T`" rule is specified by the given immediate value *val*. We capture the execution time for both rules, by t, which is 2.

4.3 Security Properties

In this section, we formally introduce the properties of Nemesis-sensitive and time-sensitive non-interference, which is the security policy that we aim to be verified by our proposed security type system.

Following prior work in the literature [3, 9, 16, 38], we assume that all components of the defined state are classified as either high or low, where the security level high denotes confidential information and low is used for public data. Formally, we assume a function $\Gamma_{i\in\{r,m,s,sr,e\}} : x \longrightarrow \{\mathbb{L}, \mathbb{H}\}$, which assigns a security level to status flags, the whole memory, registers, stack, and execution points. Then, we define a comparison operator $\simeq_\Gamma$ and set $f \simeq_\Gamma f'$ to express iff $\forall x : \Gamma(x) = \mathbb{L} \Leftrightarrow f(x) = f'(x)$, where f models sr, r, m and st. Prior work in [16] considers a program to be free of timing side channel vulnerabilities and illicit information flow if the following condition is satisfied:

Definition 4.1. A program P with starting and finishing security levels Γ and Γ', satisfies time-sensitive non-interference property if and only if:

$$\forall s_0, s_0', s_1, s_1' \in State, \forall t, t' \in \mathbb{N} :$$
$$(s_0 \simeq_\Gamma s_0') \wedge (s_0 \xrightarrow{t} s_1) \wedge (s_0' \xrightarrow{t'} s_1') \Leftrightarrow (s_1 \simeq_{\Gamma'} s_1') \wedge (t = t')$$

If a program satisfies the above definition, then it is impossible to determine the value of secret data by observing the program's overall execution time or the program's result.

To guarantee the absence of Nemesis vulnerabilities in a program we propose the nemesis-sensitive property. Information leakage through Nemesis vulnerability happens when the if/else branch contains different number of instructions or at least one instruction takes different execution time. To compare two branches of each branching instruction, we compute control dependence regions and junction points of each execution point by employing well-known techniques proposed in [8, 16]. In particular, we check the correctness of provided regions by applying safe over-approximation properties properties defined in [16]. We use two functions $region^{then}(ep)$ and $region^{else}(ep)$ to capture the set of execution points belonging to the branch target and the other region. Now assume ep^i is a branching instruction with $ep^i \in region(ep)$. Then only one branch of ep^i is executed in a real program. Hence, We observe this by selecting the execution points in the then-branch of ep^i. By typability, it is ensured that the program is free of Nemesis vulnerabilities in the secret-dependence branch ep^i. We define the nemesis-sensitive property by the following condition:

Definition 4.2. A program P with a secret-dependence branch in ep and $\text{region}^{\text{then}}$(ep) and $\text{region}^{\text{else}}$(ep) with the same number of execution points, satisfies nemesis-sensitive property if and only if:

$$\forall ep^i \in region^{then}(ep) : \forall ep^j \in region^{else}(ep) \; such \; that \; i = j :$$
$$(s_{ep^i} \xrightarrow{t} s_{ep^i_{next}}) \wedge (s_{ep^j} \xrightarrow{t'} s_{ep^j_{next}}) \Leftrightarrow t = t'$$

The above definition states if a program satisfies the nemesis-sensitive property, then an attacker cannot differentiate between secret-dependence program branches or extract sensitive data by measurement of instruction's execution time.

4.4 Security Type System

We provide a security type system to verify whether an MSP430 assembly program satisfies the above properties. In other words, a typable program ensures the absence of timing side channels, Nemesis vulnerability, and undesired information flow.

We employ the notations and definitions that were used in [16]. Thereby, we define the typability of a program P with a judgement of the form $\Gamma \vdash P(ep) : \Gamma'$. The meaning of this judgement is that, assuming we execute the INSTR in *ep* under the security level Γ and the specific constraints, then the new security level is Γ'. Some selected typing rules are presented in Figure 3. The following typing rules decompose the program into trivial judgements. Hence, we can easily impose constraints on security levels of all successive execution points and prevent timing leakages in secret-dependence branches or loops.

In the defined typing rule for instruction "`PUSH Rs`", we calculate an upper bound on the security levels of *Rs*, the stack pointer register ($SP/R1$ in MSP340) and *ep* by ($\cup$). Then we lift the security levels of the stack pointer and the top element of stack to the new security level. Other components are left unmodified by executing of *PUSH* instruction. By raising the security levels, we ensure the lack of secret information flows to attacker-observable outputs.

The instruction "`ADDC Rs, x(Rd)`" modifies the content of memory and status flags. Thereby, in this typing rule, we raise the security levels of this components to the upper bound of the security levels of the operands, the carry flag and *ep*. By lifting the security levels, we guarantee the absence of flows from high security level operands, carry, or branching conditions to an low security level sum and status flags.

We define two typing rules for the instruction "`JNZ val`" depending on the security levels of the zero flag and *ep*. In the typing rule "`JNZ-L`", we suppose the conditional jump instruction depends

$$\text{PUSH:}\ \frac{\begin{array}{c}P(ep) := PUSH\ \ Rs \\ \Gamma\text{fi}_s(TOP) := \Gamma_e(ep) \cup \Gamma_r(SP) \cup \Gamma_r(Rs) \\ \Gamma_r\text{fi}(SP) := \Gamma_e(ep) \cup \Gamma_r(SP) \cup \Gamma_r(Rs)\end{array}}{(\Gamma_{sr}, \Gamma_r, \Gamma_m, \Gamma_s) \vdash P(ep) : (\Gamma_{sr}, \Gamma\text{fi}_r, \Gamma_m, \Gamma\text{fi}_s)}$$

$$\text{ADDC:}\ \frac{\begin{array}{c}P(ep) := ADDC\ \ Rs, x(Rd) \\ \Gamma\text{fi}_m(M) := \Gamma_m(M) \cup \Gamma_r(Rs) \cup \Gamma_e(ep) \cup \Gamma_{sr}(C) \\ \Gamma\text{fi}_{sr}(C, Z, N, O) := \Gamma_m(M) \cup \Gamma_r(Rs) \cup \Gamma_e(ep) \cup \Gamma_{sr}(C)\end{array}}{(\Gamma_{sr}, \Gamma_r, \Gamma_m, \Gamma_s) \vdash P(ep) : (\Gamma\text{fi}_{sr}, \Gamma_r, \Gamma\text{fi}_m, \Gamma_s)}$$

$$\text{MOV:}\ \frac{\begin{array}{c}P(ep) := MOV\ \ @Rs,\ \ Rd \\ \Gamma\text{fi}_r(Rd) := \Gamma_m(M) \cup \Gamma_e(ep)\end{array}}{(\Gamma_{sr}, \Gamma_r, \Gamma_m, \Gamma_s) \vdash P(ep) : (\Gamma_{sr}, \Gamma\text{fi}_r, \Gamma_m, \Gamma_s)}$$

$$\text{JNZ-L:}\ \frac{\begin{array}{c}P(ep) := JNZ\ \ \ val \\ \Gamma_e(ep) \cup \Gamma_{sr}(Z) := \mathbb{L}\end{array}}{(\Gamma_{sr}, \Gamma_r, \Gamma_m, \Gamma_s) \vdash P(ep) : (\Gamma_{sr}, \Gamma_r, \Gamma_m, \Gamma_s)}$$

$$\text{JNZ-H:}\ \frac{\begin{array}{c}P(ep) := JNZ\ \ \ val \\ \Gamma_e(ep) \cup \Gamma_{sr}(Z) := \mathbb{H} \\ \neg loop(ep),\ \Gamma\text{fi}_s(S) := lift(\Gamma_s, \mathbb{H}) \\ \forall ep\text{fi} \in region(ep) : \Gamma_e(ep\text{fi}) = \mathbb{H} \\ branch\_time^{then}(ep) := branch\_time^{else}(ep) \\ count(region^{then}(ep)) := count(region^{else}(ep)) \\ \forall ep^i \in region^{then}(ep), \\ \forall ep^j \in region^{else}(ep), i = j : runtime(ep^i) := runtime(ep^j)\end{array}}{(\Gamma_{sr}, \Gamma_r, \Gamma_m, \Gamma_s) \vdash P(ep) : (\Gamma_{sr}, \Gamma_r, \Gamma_m, \Gamma\text{fi}_s)}$$

Figure 3: Selected typing rules.

on the information with low security level. A conditional jump instruction does not change any information. Accordingly, in this typing rule, any security levels are not modified. The typing rule "JNZ−H" is defined for secret-dependent jump instructions. We consider the following constraints for this typing rule to ensure the absence of information leakage. (1) The secret-dependent loops are forbidden to make the number of iterations invisible for attacker. (2) The security level of the stack are raised to high. (3) All execution points belonging to branching instruction are assigned a high security level. (4) The execution time required for the then-branch should be equal to the execution time of the else-branch. (5) Each branch must have the same number of MSP430 instruction. (6) The corresponding instructions in each branch must have the same execution time.

5 SCF^{MSP}: DESIGN AND IMPLEMENTATION

In this section, we discuss the design and implementation of SCF^{MSP}, our static analysis tool for evaluating the security of MSP430 assembly programs against side channel attacks. Side channel evaluations are usually costly and time consuming. Hence, we develop SCF^{MSP} for verifying side channel vulnerabilities in a fast, inexpensive, and automated manner based on our proposed security type system. We make the tool and evaluation benchmarks publicly available at https://github.com/sepidehpouyan/SCF-MSP430.

A high-level workflow diagram of SCF^{MSP} tool and its components is illustrated in Figure 4. The gray box represents the primary

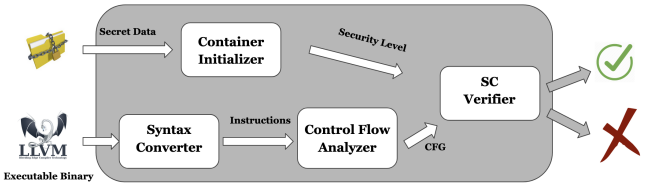

Figure 4: Workflow of SCF^{MSP} tool.

parts of this tool which implements our proposed evaluation technique for side channel detection. The SCF^{MSP} tool is implemented in roughly 1500 lines of python code and leverages multiple existing python libraries which are described later. The tool currently reports violations of our security properties (cf. Section 4) in three classes of vulnerabilities. Intuitively these are defined as:

- **Timing leaks:** if a program takes different execution time based on the secret input. By measuring and comparing start-to-end running times for if/else bodies in the secret-dependence branches, SCF^{MSP} is able to detect timing leakage in an assembly program.
- **Nemesis vulnerability:** if a program takes a different execution time for at least one instruction in the if/else branch. The SCF^{MSP} is no longer restricted to start-to-end timing measurements.
- **Information leakage:** if there is an undesired information flow from secret inputs to observable output.

We now give a brief overview of each of these components and their implementation:

Container Initialiser. Given a list of starting function's arguments from high-level code in JSON format. This component assigns a security level to each register and memory. We associate the arguments and their corresponding registers according to the MSP430 calling conventions [23].

Syntax Converter. We employ this component to convert the input ELF files into a form close to our defined syntax in Section 4. The ELF files are typically the output of a compiler or linker and are a binary format of programs that defines the structure for binaries, libraries, and core files. We use the python library *pyelftools* [37] to reach the starting point of instructions in the ELF file. Afterwards, we parse instructions based on the MSP430 instruction format illustrated in Figure 5. More specifically, we extract opcodes, addressing modes, and operands of each instruction (Figure 6). Then, we compute the length of each instruction depending on the addressing mode to find the next instruction.

Next, the extracted data is passed to the objects which specify attributes and functionality of related to each instruction. Specifically, the required clock cycle, the subsequent execution points, and the corresponding typing rules is determined to check the effect of each instruction in the information flow and the running time of the program.

Control Flow Analysis. The primary goal of using this component is to build a precise control flow graph of the MSP430 assembly programs by leveraging the python library *NetworkX* [21]. The resulting graph is used to identify all successors and predecessors

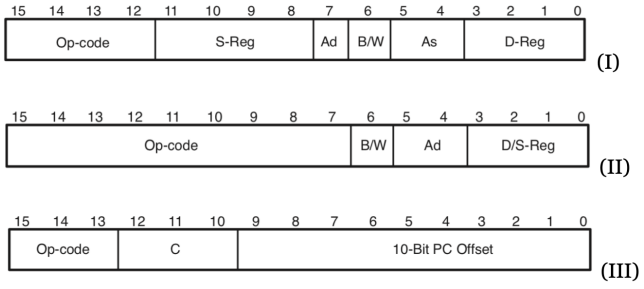

Figure 5: (I) Double operand, (II) Single operand, (III) Jump Instruction Format [22].

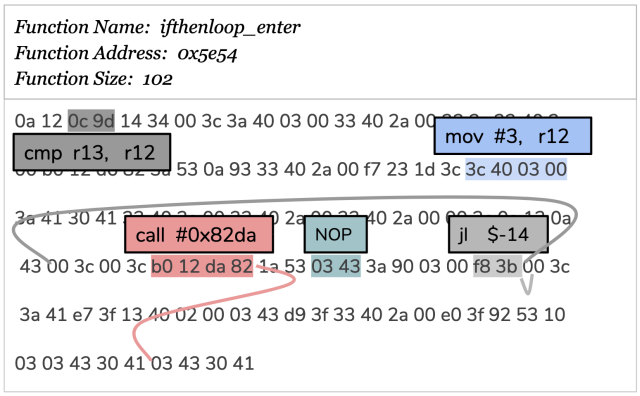

Figure 6: Instruction Parsing.

of an execution point to achieve two goals: (1) to capture the next point that needs to be analysed, (2) to restrict the information flow of consecutive execution points. Further, we obtain the immediate dominators and post-dominators of all execution points to compute control dependence regions of branches and as well as, to infer loops in an assembly program.

SC Verifier. We send the outputs of previous components to the SC Verifier to conduct a series of static analysis for identifying side channel leaks in the MSP430 assembly programs. Toward this end, we propagate secret assignments from user-annotated secret inputs to the used containers during executing program according to the typing rule of the current analysing instruction. It checks the Nemesis-type vulnerability in a secret-dependence branching instruction by comparing the execution time for each pair of corresponding instructions in the if/else regions. Furthermore, the potential timing leaks in unbalanced conditional jumps is detected by calculating the overall running time of each if/else regions. We perform this type checking for all program's instructions in a loop iteration, and finally report the results of our static verifying to the user. If there is a detection during type checking, we provide an error message that identifies the vulnerability type (timing, Nemesis, Information leak and also where a given attack occurred (i.e., address), so a programmer could fix identified vulnerabilities.

We adapt parts of the open source code provided for the SCF^{AVR} tool [16]. More specifically, we customise the SCF^{AVR} to implement our defined typing rules for MSP430 instructions and to consider different addressing modes and constant generators in the pre-computations. Furthermore, We extend the static analysis to check for Nemesis-style vulnerability. However, in contrast to the SCF^{AVR}, we parse the ELF files instead of object dump files to cover all core and emulated MSP430 instructions, to be independent of the structure of the code generated by the compiler, and to address a range of issues of the `msp430-objdump` disassembler.

6 EVALUATION

In this section, we describe our evaluation of SCF^{MSP} to demonstrate the accuracy of our static analysis in side channels detection.

We have evaluated SCF^{MSP} based on a set of hand-crafted benchmark containing vulnerable and benign programs that expose typical and challenging patterns in secret-dependent control flow. The benchmark consists of 16 vulnerable C programs with annotations that identify secret control variables. The programs are being compiled once with the off-the-shelf LLVM backend for the MSP430, resulting in a vulnerable binary programme. In addition, a second version of the assembly code is produced, where instructions in secret-dependent branches are balanced out with respect to the individual instructions' execution times (loosely following [3]). On the MSP430 architecture, this approach should be equivalent to following the constant-time policy [4, 7, 11], and results in semantically equivalent programs without side-channels leakage. Below we give a short explanation of some of these benchmark programs:

- *diamond*: the unsafe version of this program contains two secret-dependent branching instructions and one secret-independent branch. In the safe version some dummy instructions are added to the binary code to balance two branches.
- *loop*: this program contains a loop which has a public data in it's condition and a secret-dependent branch in the loop-body. In the safe version, the branch is balanced to execute at the same time.
- *multifork*: this program contains a switch statement that tests the value of a secret variable and then compares it with multiple cases. In the safe version, each branch is executed at the same time.
- *call*: in this program a function is called in a secret-dependent branch body. Our analysis with SCF^{MSP} detects timing leak that is removed in the safe version.
- *ifthenloop*: this program contains a branch depending on a secret data, and a loop with public variable in its condition is executed if the branch was taken. Based on our defined type system, all execution points in the body of the loop should be tagged as secret points.
- *ifthenlooplooptail*: this program contains nested secret-dependent branches as well as nested loops to test SCF^{MSP} in more complex control flows.

In Table 3, we detail experimental results from applying SCF^{MSP} on the benchmark. Our results indicate that SCF^{MSP} is capable of analysing reasonable complex MSP430 programs, and identifies a range of side-channel vulnerabilities and information declassification efficiently. Note that a range of our examples exhibit information leakage in the "unsafe" and "safe" cases. This is because branch balancing or constant-time programming cannot eliminate

Table 3: Evaluation results for SCF^{MSP} on the benchmark suite. For each compiled example program we give an indication for C program complexity (LOC = lines of code, CFG Size = number of nodes in the program's Control Flow Graph), execution time of SCF^{MSP}, and list the vulnerabilities found by SCF^{MSP}.

Benchmark	Version	LOC	CFG Size	SCF^{MSP} exec time	Timing Leak	Nemesis vulnerable	Information Leak
diamond	Unsafe	22	14	0.419s	✓	✓	✓
diamond	Safe	22	28	0.438s	✗	✗	✓
loop	Unsafe	15	38	0.942s	✓	✓	✓
loop	Safe	15	45	0.948s	✗	✗	✓
fork	Unsafe	9	6	0.412s	✓	✓	✓
fork	Safe	9	11	0.443s	✗	✗	✓
indirect	Unsafe	13	9	0.422s	✓	✓	✓
indirect	Safe	13	23	0.465s	✗	✗	✓
multifork	Unsafe	14	14	0.426s	✓	✓	✓
multifork	Safe	14	30	0.496s	✗	✗	✓
call	Unsafe	7	8	0.434s	✓	✓	✓
call	Safe	7	19	0.478s	✗	✗	✓
ifcompound	Unsafe	9	28	0.952s	✓	✓	✓
ifcompound	Safe	9	36	0.982s	✗	✗	✓
ifthenloop	Unsafe	10	13	0.437s	✓	✓	✗
ifthenloop	Safe	10	42	0.505s	✗	✗	✗
ifthenloopif	Unsafe	12	29	0.446s	✓	✓	✗
ifthenloopif	Safe	12	92	0.628s	✗	✗	✗
ifthenlooploop	Unsafe	12	25	0.444s	✓	✓	✗
ifthenlooploop	Safe	12	70	0.522s	✗	✗	✗
ifthenlooplooptail	Unsafe	17	44	0.499s	✓	✓	✗
ifthenlooplooptail	Safe	17	118	0.734s	✗	✗	✗
triangle	Unsafe	5	6	0.412s	✓	✓	✓
triangle	Safe	5	13	0.438s	✗	✗	✓
multiply	Safe	3	15	0.907s	✗	✗	✓
mulhi	Unsafe	15	41	0.420s	✓	✓	✓
mulhi	Safe	15	55	0.589s	✗	✗	✓
modexp	Unsafe	13	97	4.178s	✓	✓	✓
modexp	Safe	13	116	2.409s	✗	✗	✓
bsl	Unsafe	20	15	0.440s	✓	✓	✗
bsl	Safe	20	29	0.465s	✗	✗	✗

these vulnerabilities, and because of a lack of precision in our static analysis w.r.t. determining reaching definitions for register- and memory cell assignments. Extending the implementation towards dealing with the static analysis limitations will help to improve the precision of the analysis in the future, and allow us to correctly track information flow beyond register content and across protection domains provided by Sancus [33].

7 CONCLUSIONS AND FUTURE DIRECTIONS

This paper proposed a methodology to detect microarchitectural side channels including Nemesis-type interrupt latency attacks, timing side channels and other possibilities for information leakage in MSP430 assembly programs. We provided a complete small-step operational semantics for MSP430 instructions, a security type system with some defined security properties, and a static analysis tool to automatically check MSP430 programs against side channels based on our security type system. Since the execution time of programs in MSP430 microcontrollers is deterministic, our approach verifies the absence of side channels more effective than previous works.

We have evaluated our tool by applying it on a number of self-implemented vulnerable and benign MSP430 assembly programs and have shown that it can find a range of side channels in the widely used and complicated structures in programs, as well as it can verify that the repaired versions of vulnerable programs do not exhibit the original vulnerability.

There are a number of directions for future work. First, we would like to improve the accuracy of the SCF^{MSP} by providing symbolic execution-based analysis to have fine-grained control over the memory. Like any other program static analysis tool, the SCF^{MSP} has no insight into the register content or where in memory the pointed data is stored. More specifically, the whole memory is annotated with a single security level and only immediate value is accepted for call instructions. Hence, the SCF^{MSP} is incomplete and may report false positives. Second, we plan to evaluate the SCF^{MSP} broadly

by applying it on the real-world security critical systems. A starting point for this could be a comprehensive evaluation based on a realistic case study such as Vulcan [13] or implementations of cryptographic functions.

Acknowledgments. This research is partially funded by the Research Fund KU Leuven. We thank Hans Winderix for supporting this research and for making his benchmarks available to us.

REFERENCES

[1] Hezam Akram Abdul-Ghani, Dimitri Konstantas, and Mohammed Mahyoub. 2018. A Comprehensive IoT Attacks Survey based on a Building-blocked Reference Mode. *International Journal of Advanced Computer Science and Applications* 9, 3 (2018). https://doi.org/10.14569/IJACSA.2018.090349

[2] Onur Acıiçmez, Çetin Koç, and Jean-Pierre Seifert. 2007. Predicting Secret Keys Via Branch Prediction. In *CryptographersâĂŹ Track at the RSA Conference*. Springer, 225–242. https://doi.org/10.1007/11967668_15

[3] Johan Agat. 2000. Transforming out Timing Leaks. In *Proceeding of the 27th ACM Symposium on Principles of Programming Languages(POPL)*. ACM Press, 40–53. https://doi.org/10.1145/325694.325702

[4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying constant-time implementations. In *25th USENIX Security Symposium (USENIX Security 16)*. 53–70.

[5] Jude Angelo Ambrose, Roshan G. Ragel, Darshana Jayasinghe, Tuo Li, and Sri Parameswaran. 2015. Side channel attacks in embedded systems: A tale of hostilities and deterrence. *16th. International Symposium on Quality Electronic Design* (2015), 452–459. https://doi.org/10.1109/ISQED.2015.7085468

[6] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. 2014. System-level non-interference for constant-time cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 1267–1279.

[7] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. 2018. Secure compilation of side-channel countermeasures: the case of cryptographic âĂIJconstant-timeâĂİ. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 328–343.

[8] Gilles Barthe and Tamara Rezk. 2005. Non-interference for a JVM-like language. In *Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*. 103–112.

[9] Gilles Barthe, Tamara Rezk, and Martijn Warnier. 2006. Preventing Timing Leaks Through Transactional Branching Instructions. *Electronic Notes in Theoretical Computer Science* 153 (May 2006), 33–55. https://doi.org/10.1016/j.entcs.2005.10.031

[10] Daniel J. Bernstein. 2005. Cache-timing attacks on AES. (2005). http://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[11] Daniel J Bernstein, Tanja Lange, and Peter Schwabe. 2012. The security impact of a new cryptographic library. In *International Conference on Cryptology and Information Security in Latin America*. Springer, 159–176.

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

[13] Jo Van Bulck, Jan Tobias Mühlberg, and Frank Piessens. 2017. VulCAN: Efficient component authentication and software isolation for automotive control networks. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, Vol. Part F132521. San Juan, Puerto Rico, 225–237.

[14] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2018. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *CCS '18*.

[15] Jeroen V Cleemput, Bart Coppens, and Bjorn De Sutter. 2012. Compiler mitigations for time attacks on modern x86 processors. *ACM Transactions on Architecture and Code Optimization (TACO)* 8, 4 (2012), 1–20.

[16] Florian Dewald, Heiko Mantel, and Alexandra Weber. 2017. AVR processors as a platform for language-based security. In *Computer Security âĂŞ ESORICS 2017*. Springer, 427–445. https://doi.org/10.1007/978-3-319-66402-6_25

[17] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2015. Cacheaudit: A tool for the static analysis of cache side channels. *ACM Transactions on Information and System Security (TISSEC)* 18, 1 (2015), 1–32.

[18] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering* 8, 1 (2018), 1–27. https://doi.org/10.1007/s13389-016-0141-6

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

[20] Travis Goodspeed. 2008. Practical Attacks against the MSP 430 BSL. In *Twenty-Fifth Chaos Communications Congress*. Berlin, Germany.

[21] Aric Hagberg, Pieter Swart, and Daniel S. Chult. 2008. *Exploring network structure, dynamics, and function using NetworkX*. Technical Report. 11–15 pages.

[22] Texas Instruments. 2006. Msp430x1xx family userâĂŹs guide. (2006). http://www.ti.com/lit/ug/slau049f/slau049f.pdf

[23] Texas Instruments. 2013. The MSP430 Embedded Application Binary Interface. (2013). http://www.ti.com/lit/an/slaa534/slaa534.pdf

[24] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–19.

[25] Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential Power Analysis. In *Advances in Cryptology — CRYPTO' 99*, Michael Wiener (Ed.), Vol. 1666. Springer Berlin Heidelberg, Berlin, Heidelberg, 388–397. https://doi.org/10.1007/3-540-48405-1_25

[26] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology — CRYPTO '96*, Neal Koblitz (Ed.), Vol. 1109. Springer Berlin Heidelberg, Berlin, Heidelberg, 104–113. https://doi.org/10.1007/3-540-68697-5_9

[27] Boris Köpf and Heiko Mantel. 2007. Transformational typing and unification for automatically correcting insecure programs. *International Journal of Information Security* 6, 2-3 (2007), 107–131. https://doi.org/10.1007/s10207-007-0016-z

[28] Xun Li, Vineeth Kashyap, Jason K Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T Chong. 2014. Sapper: A language for hardware-level security policy enforcement. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. 97–112.

[29] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading kernel memory from user space. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 973–990.

[30] Heiko Mantel, Johannes Schickel, Alexandra Weber, and Friedrich Weber. 2018. How Secure Is Green IT? The Case of Software-Based Energy Side Channels. In *Proceedings of the 23rd European Symposium on Research in Computer Security (ESORICS)*, Javier Lopez, Jianying Zhou, and Miguel Soriano (Eds.). Springer International Publishing, Barcelona, Spain, 218–239. https://doi.org/10.1007/978-3-319-99073-6_11

[31] Heiko Mantel and Artem Starostin. 2015. Transforming out timing leaks, more or less. In *Computer Security - ESORICS 2015*, Vol. 9326. Springer, 447–467.

[32] Amir Moradi and Gesine Hinterwälder. 2015. Side-Channel Security Analysis of Ultra-Low-Power FRAM-Based MCUs. In *6th International Workshop on Constructive Side-Channel Analysis and Secure Design*, Stefan Mangard and Axel Y. Poschmann (Eds.), Vol. 9064. Springer International Publishing, Cham, 239–254. https://doi.org/10.1007/978-3-319-21476-4_16

[33] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. 2013. Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base. In *22nd USENIX Security Symposium*. 479âĂŞ494.

[34] Job Noorman, Felix Freiling, Jo Van Bulck, Jan Tobias MÃijhlberg, Frank Piessens, Pieter Maene, Bart Preneel, Ingrid Verbauwhede, Johannes GÃűtzfried, and Tilo MÃijller. 2017. Sancus 2.0: A low-cost security architecture for IoT devices. *ACM Transactions on Privacy and Security (TOPS)* 20, 3, Article 7 (September 2017), 7:1-7:33 pages. https://doi.org/10.1145/3079763

[35] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2005. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology - CT-RSA 2006, The CryptographersâĂŹ Track at the RSA Conference 2006*. Springer-Verlag, 1âĂŞ20. https://doi.org/10.1007/11605805_1

[36] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *CryptographersâĂŹ track at the RSA conference*. Springer, 1–20.

[37] pyelftools. 2011. pyelftools: Parsing ELF and DWARF in Python. (2011). Retrieved 5 Dec 2019 from https://github.com/eliben/pyelftools

[38] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE Journal on selected areas in communications* 21, 1 (2003), 5–19.

[39] Samuel Tweneboah-Koduah, Knud Erik Skouby, and Reza Tadayon. 2017. Cyber Security Threats to IoT Applications and Service Domains. *Wireless Personal Communications* 95, 1 (2017), 169–185. https://doi.org/10.1007/s11277-017-4434-6

[40] Dennis Volpano and Geoffrey Smith. 1997. Eliminating covert flows with minimum typings. In *Proceedings 10th Computer Security Foundations Workshop*. IEEE, 156–168. https://doi.org/10.1109/CSFW.1997.596807

[41] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. 2018. MicroWalk: A framework for finding side channels in binaries. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 161–173.

[42] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 719–732.

[43] Danfeng Zhang, Yao Wang, Edward Suh, and Andrew C. Myers. 2015. A hardware design language for timing-sensitive information-flow security. *Acm Sigplan Notices* 50, 4 (2015), 503–516.