AVR Processors as a Platform for Language-Based Security

Florian Dewald, Heiko Mantel, and Alexandra Weber

Computer Science Department, TU Darmstadt, Germany
{dewald,mantel,weber}@mais.informatik.tu-darmstadt.de

Abstract AVR processors are widely used in embedded devices. Hence, it is crucial for the security of such devices that cryptography on AVR processors is implemented securely. Timing-side-channel vulnerabilities and other possibilities for information leakage pose serious dangers to the security of cryptographic implementations. In this article, we propose a framework for verifying that AVR assembly programs are free from such vulnerabilities. In the construction of our framework, we exploit specifics of the 8-bit AVR architecture to make the static analysis of timing behavior reliable. We prove the soundness of our analysis against a formalization of the official AVR instruction-set specification.

1 Introduction

AVR processors are popular microcontrollers for embedded devices [45]. These processors are used, for instance, in the Internet of Things [47]. There are also specialized AVR processors by Atmel for aerospace [8] and automotive [7] applications. Hence, AVR processors are an attractive target for attacks.

Cryptographic implementations for AVR microcontrollers are available directly in hardware [4] and also in software. Cryptographic libraries for AVR include, for instance, μNaCl [29], AVR-Crypto-Lib [19], and TinyECC [35]. The current versions of these libraries differ in the level of security they provide against side channels. For instance, the library μNaCl was developed with a focus on avoiding side-channel vulnerabilities [29] while AVR-Crypto-Lib so far does not contain protection mechanisms against side-channel attacks [19].

Hardware implementations of cryptography on AVR microcontrollers have been attacked successfully through side-channel attacks [30, 43]. Recently, Ronen, O'Flynn, Shamir and Weingarten [47] mounted a side-channel attack based on power consumption on smart light bulbs that contain the Atmel ATmega2564RFR2 System on Chip. The attack exploited that the power consumption of an AES encryption on the AVR microcontroller depends on the secret AES key. Ronen, O'Flynn, Shamir and Weingarten recovered the entire key and used it to authenticate compromised firmware for the smart light bulbs.

Side-channel attacks can be based on a multitude of execution characteristics like cache behavior [36, 44], power consumption [32, 47] or running time [24, 31]. Attacks that exploit the running time of an execution are particularly dangerous

Published in:
S. N. Foley et al. (Eds.): ESORICS 2017, Part I, LNCS 10492, pp. 427–445, 2017.
© Springer International Publishing AG 2017
The final publication is available at Springer via https://dx.doi.org/10.1007/978-3-319-66402-6_25

because they can be mounted remotely without physical access to a system [16, 17]. In this article, we focus on such timing side channels.

Language-based techniques for detecting and mitigating timing side channels exist for multiple programming languages [2,11,33,41,49]. However, the models of time underlying the soundness proofs for these techniques do not capture optimizations like caches or branch prediction faithfully. As a consequence, the soundness proofs for these techniques are less effective in practice than one might expect, e.g., on x86 processors [40]. On 8-bit AVR microcontrollers, the time required to execute an instruction can be predicted statically. This is the feature of AVR processors that we exploit in this article.

Based on the predictability of execution times, we propose a security type system for AVR assembly. Our type system reliably verifies that there are no possibilities for information leakage in a timing-sensitive and flow-sensitive fashion. We base our soundness proof on a formal operational semantics of AVR assembly that reflects the execution times specified in the AVR instruction set manual [6]. Building on our security type system, we developed the Side-Channel Finder$^{\text{AVR}}$ (SCF$^{\text{AVR}}$), a tool for checking AVR assembly programs against timing-side-channel vulnerabilities and other possibilities for information leakage.

We show that our type system can be used to check realistic programs by applying SCF$^{\text{AVR}}$ to the implementations of the stream cipher Salsa20 and to the Message-Authentication Code Poly1305 from the library μNaCl. To prove the type system's soundness, we developed a formal semantics for AVR assembly, because none was available so far. We make our semantics available to others,[1] such that they can use it for proving the soundness of program analyses for AVR.

2 Preliminaries

2.1 Timing-Side-Channel Vulnerabilities and Attacker Models

Timing-Side-Channel Vulnerabilities. Consider the following example program with secret information stored in variable `h`.

```
if (h = 1) then sleep(1000) else skip;
```

If the variable `h` has value 1, the *then*-branch will be executed, and the program will sleep for 1000 milliseconds. If the variable `h` has a value other than 1, then the *else*-branch will be executed, and the overall execution will be faster in this second case. Such a dependency of a program's execution time on secret information is called a *timing-side-channel vulnerability*. If an attacker can observe the execution time of a program, then he can, indeed, exploit such vulnerabilities to deduce critical secrets (as shown, e.g., in [31]).

Attacker Models. An attacker model defines what an attacker can observe during a program execution. We consider a passive attacker who has knowledge of the program's code and can observe execution time as well as certain inputs and outputs. There are multiple possibilities to define attacker models. In this article,

[1] The addendum to this article and the tool SCF$^{\text{AVR}}$ are available under `http://www.mais.informatik.tu-darmstadt.de/scf2017.html`.

we model the visibility of information containers for an attacker by the security levels $\mathcal{L}$ (visible) and $\mathcal{H}$ (secret and invisible to the attacker), and we assign one level to each input (initial state of registers, etc.) and each output (final state of registers, etc.) of a program. We call such an assignment of security levels to information containers a *domain assignment*. We call two given states *indistinguishable to an attacker under a domain assignment* if these states assign identical values to each container labeled with $\mathcal{L}$.

2.2 Static Analysis

Timing-Sensitive Information-Flow Analysis. An information-flow analysis checks for the absence of undesired information flow in a program. The resulting security guarantee is usually captured by a variant of *noninterference* [26], i.e., by a formally defined security property that requires secret information to not influence the observations of an attacker. The choice of an execution model and an attacker model influences which variant of noninterference is suitable [39]. Research on information-flow analyses goes back to Denning and Denning [20,21] and Cohen [18]. A comprehensive survey of language-based information-flow analyses has been provided by Sabelfeld and Myers in [48].

Information-flow analyses usually over-approximate the flow of secret information to attacker-observable outputs. There are multiple approaches to analyzing information-flow security. In this article, we focus on security type systems. A security type system formalizes constraints on the sensitivity of data stored in containers (e.g., in registers) during the execution of a program. If a program satisfies these constraints for a domain assignment, then the program is called *typable under the domain assignment*. A type system is *sound* with respect to a security property if and only if all programs that are typable under some domain assignment satisfy the security property under this domain assignment.

A timing-sensitive property takes the influence of secrets on the running time of a program into account. The semantics on which a timing-sensitive security property is based should, hence, capture the execution time of the program sufficiently precisely. A timing-sensitive information flow analysis tries to anticipate such dependences between running times and secrets (see, e.g., [2,49]).

Control Flow Analysis. Assembly languages have unstructured control flow. To determine the control flow of AVR assembly code, we employ the approach and notation that was proposed in [10] and has inspired many others (e.g., [37]). In particular, we define the control-dependence region and junction point of each program point using Safe Over Approximation Properties (SOAPs).

To distinguish branchings from loops, we base on the concept of natural loops [3, Chapter 18.1]. Natural loops are defined based on the notions of domination introduced by Prosser [46] and back edges in control flow graphs. A node n_1 in a control flow graph dominates a node n_2, written $n_1 \;\mathsf{dom}\; n_2$, if and only if all paths from the root to n_2 go trough n_1. An edge from node n_2 to node n_1 in the control flow graph of a program is a *back edge* if and only if $n_1 \;\mathsf{dom}\; n_2$. The natural loop of a back edge from n_2 to n_1 contains all execution points that are dominated by n_1 and from which n_2 is reachable without passing n_1.

2.3 AVR Assembly Instruction Set

The Atmel AVR 8-bit instruction set consists of 119 distinct instructions. The instructions operate on memory, registers, and a stack. A dedicated status register stores status flags, e.g., the carry flag indicating whether the most recently executed instruction resulted in a carry.

Although 8-bit AVR microprocessors are widely used, they do not support caching and branch prediction. Memory accesses take only one clock cycle, which makes caches dispensable [34]. Most instructions are executed in one fixed number of clock cycles on 8-bit AVR processors. However, for conditional jumps, two fixed execution times are possible, depending on the outcome of the branching condition. If a jump is performed, then the instruction takes an additional clock cycle. The behavior and execution time of the individual AVR instructions are defined informally in the instruction set manual [6]. This description constitutes the basis for our formalization of the semantics in Section 3.

2.4 Notation

We denote the i-th bit of the binary representation of $v \in \mathbb{Z}$ by $v_{[i]}$. Given a function r, we write $r[x{\mapsto}y]$ for the function resulting from updating r at x with y. We use this notation also, if y is one bit too long with respect to $\mathtt{rng}(r)$. In this case, we define the update by $r[x{\mapsto}y](x) = y'$ where y' results from y by dropping the most significant bit in the binary representation. For Boolean values, we define the notation $r[x \mapsto_s \mathit{True}] := r[x \mapsto 1]$ and $r[x \mapsto_s \mathit{False}] := r[x \mapsto 0]$.

3 Our Formal Semantics of AVR Assembly Programs

We show how to exploit the predictability of execution times on AVR processors to obtain a faithful reference point for a sound security analysis. To this end, we define a formal operational semantics for AVR assembly code based on [6].

3.1 Syntax

In AVR assembly, instructions are represented by mnemonics, i.e., keywords that describe the purpose of the instruction. The mnemonics also determine the number and types of the arguments in an instruction.

We define the syntax of AVR assembly instructions by the following grammar:

$$\begin{aligned}\mathtt{INSTR} := {} & \mathit{Simple} \mid \mathit{Unary}\ \mathit{Rd} \mid \mathit{Binary}\ \mathit{Rd}\ \mathit{Rr} \mid \mathit{Control}\ \mathit{epa} \mid \mathit{Immediate}\ \mathit{Rd}\ k \mid \\ & \mathtt{out}\ k\ \mathit{Rr} \mid \mathtt{ld}\ \mathit{Rd}\ \mathit{Rs}\ * \mid \mathtt{st}\ \mathit{Rs}\ \mathit{Rr}\ * \mid \mathtt{ldd}\ \mathit{Rd}\ \mathit{Rs}\ k \mid \mathtt{std}\ \mathit{Rs}\ \mathit{Rr}\ k\end{aligned}$$

where $\mathit{Simple} \in \{\mathtt{clc}, \mathtt{cli}, \mathtt{ret}\}$, $\mathit{Unary} \in \{\mathtt{dec}, \mathtt{inc}, \mathtt{lsr}, \mathtt{neg}, \mathtt{pop}, \mathtt{push}, \mathtt{ror}\}$, $\mathit{Binary} \in \{\mathtt{adc}, \mathtt{add}, \mathtt{and}, \mathtt{cp}, \mathtt{cpc}, \mathtt{cpse}, \mathtt{eor}, \mathtt{mov}, \mathtt{movw}, \mathtt{mul}, \mathtt{or}, \mathtt{sbc}, \mathtt{sub}\}$, $\mathit{Control} \in \{\mathtt{brcc}, \mathtt{brcs}, \mathtt{breq}, \mathtt{brne}, \mathtt{call}, \mathtt{jmp}, \mathtt{rcall}, \mathtt{rjmp}\}$, and $\mathit{Immediate} \in \{\mathtt{adiw}, \mathtt{andi}, \mathtt{cpi}, \mathtt{in}, \mathtt{ldi}, \mathtt{sbci}, \mathtt{sbiw}, \mathtt{subi}\}$.

Each instruction consists of a mnemonic followed by at most three arguments. The arguments can be basic execution points (*epa* in the grammar above), registers (Rd, Rr, Rs), immediate values (k) or modifiers refining the behavior of I/O instructions ($*$). We define the set of basic execution points by $\texttt{EPS}_0 := \{(f, a) \mid f \in \texttt{FUNC} \land a \in \mathbb{N}\}$ where FUNC models the set of all function identifiers (e.g., labels based on source-level function names). We define the set of 8-bit registers by $\texttt{REG} := \{r_n \mid n \in [0, 31]\} \cup \{\texttt{sp}_l, \texttt{sp}_u\}$, where $\texttt{sp}_l$ and $\texttt{sp}_u$ are special registers that store the lower and the upper part of the stack pointer, respectively. To obtain 16-bit values, two registers can be used as a register pair. One common use of register pairs is to store memory addresses in the pair r_{27} and r_{26}, the pair r_{29} and r_{28}, or the pair r_{31} and r_{30}. These register pairs are commonly referred to as X, Y, and Z, respectively. We reflect this in the syntax by the set $\{X, Y, Z\}$ of special (16 bit) registers where X captures the register pair r_{27} and r_{26}, Y captures the register pair r_{29} and r_{28}, and Z captures the register pair r_{31} and r_{30}. We define the set of immediate values as $\mathbb{Z}$ and the set of modifiers for I/O instructions by $\{+, -, \#\}$.

We use the meta variable *epa* to range over $\texttt{EPS}_0$, the meta variables Rd and Rr to range over REG, the meta variable Rs to range over $\{X, Y, Z\}$, the meta variable k to range over $\mathbb{Z}$, and the meta variable $*$ to range over $\{+, -, \#\}$.

A program from the set $\texttt{PROG} := \texttt{EPS}_0 \rightharpoonup \texttt{INSTR}$ of all AVR assembly programs is modeled as a mapping from basic execution points to instructions. We only consider programs that satisfy a well-formedness criterion. We define the well-formedness of programs as the conjunction of three requirements. Firstly, we require each function to contain a unique return instruction `ret`. Secondly, we require the arguments of all instructions to lie within the ranges specified in [6] (e.g., register arguments for `adiw` and `sbiw` must be from the set $\{r_n \mid n \in \{24, 26, 28, 30\}\}$). Thirdly, we require that the immediate arguments to all `in` and `out` instructions are from the set $\{0x3f, 0x3e, 0x3d\}$, i.e., the addresses of the status register, $\texttt{sp}_u$, and $\texttt{sp}_l$ on an ATmega microcontroller [5].

In practice, valid arguments are ensured by correct compilers. All programs we encountered, e.g., in our case study on μNaCl, had a unique return instruction. For programs with multiple return instructions, a unique return instruction can be achieved by simple program rewriting.

3.2 Semantics

Our operational semantics is a small-step semantics at the granularity of AVR instructions. We include timing information by annotating transitions between execution states with the required number of clock cycles.

In our semantics, we use a function $\texttt{t} : \texttt{INSTR} \to \mathbb{N}$ to capture the fixed amount of clock cycles that each given instruction takes to execute. The definition of this function depends on the particular AVR processor. In Table 1, we define t for ATmega microcontrollers with 16 bit PC based on the timing information in [6].

To model the states during the execution of a program on an 8-bit AVR microcontroller, we define the set of values that can be represented in 8-bit two's complement notation as $\texttt{VAL}_8 := [-2^7, 2^7 - 1]$. Furthermore, we define the set ADDR of all addresses in the memory by $\texttt{ADDR} := [0, \texttt{MAXADDR}]$. We model the contents of the registers by $\texttt{REG-VAL} := \texttt{REG} \to \texttt{VAL}_8$ and the contents of the

Table 1. Instructions i grouped by required clock cycles $\mathtt{t}(i)$

$\mathtt{t}(i)$	i
1	`adc` $Rd\ Rr$, `add` $Rd\ Rr$, `and` $Rd\ Rr$, `andi` $Rd\ k$, `brcc` epa, `brcs` epa, `breq` epa, `brne` epa, `clc`, `cli`, `cp` $Rd\ Rr$, `cpc` $Rd\ Rr$, `cpse` $Rd\ Rr$, `cpi` $Rd\ k$, `dec` Rd, `eor` $Rd\ Rr$, `in` $Rd\ k$, `inc` Rd, `ld` $Rd\ Rs\ \#$, `ldi` $Rd\ k$, `lsr` Rd, `mov` $Rd\ Rr$, `movw` $Rd\ Rr$, `neg` Rd, `or` $Rd\ Rr$, `out` $k\ Rr$, `ror` Rd, `sbc` $Rd\ Rr$, `sbci` $Rd\ k$, `sub` $Rd\ Rr$, `subi` $Rd\ k$
2	`adiw` $Rd\ k$, `ld` $Rd\ Rs\ +$, `ldd` $Rd\ Rs\ k$, `mul` $Rd\ Rr$, `pop` Rd, `push` Rr, `rjmp` epa, `sbiw` $Rd\ k$, `st` $Rs\ k\ -$, `st` $Rs\ k\ +$, `st` $Rs\ k\ \#$, `std` $Rs\ Rr\ k$
3	`jmp` epa, `ld` $Rd\ Rs\ -$, `rcall` epa
4	`call` epa, `ret`

memory by $\mathtt{MEM\text{-}VAL} := \mathtt{ADDR} \to \mathtt{VAL}_8$. We model the contents of the stack as a list of 8-bit values from the set $\mathtt{STACK\text{-}VAL} := \mathtt{VAL}_8^*$, where the head of the list represents the top-most element on the stack. Like x86 processors, AVR microcontrollers use a dedicated register to store status flags. We model the state of the carry flag and the zero flag by $\mathtt{STAT\text{-}VAL} := \{C, Z\} \to \{0, 1\}$, where 0 captures that a flag is not set and 1 captures that a flag is set.

We model the program counter and the call stack by $\mathtt{EPS} := \mathtt{EPS}_0 \times \mathtt{EPS}_0^*$. We call elements of EPS execution points. In an execution point $((f, a), fs)$, fs models the call stack, and address a in function f models the program counter. A program terminates if `ret` is executed with an empty call stack. We model termination by ϵ. We define the set of possible execution states by $\mathtt{STATE} := \mathtt{STAT\text{-}VAL} \times \mathtt{MEM\text{-}VAL} \times \mathtt{REG\text{-}VAL} \times \mathtt{STACK\text{-}VAL} \times (\mathtt{EPS} \cup \{\epsilon\})$. We define the selector $\mathsf{epselect} : \mathtt{STATE} \to (\mathtt{EPS} \cup \{\epsilon\})$ to return the execution point of a given state. Furthermore, we define the addition of a number to an execution point by $((f, a), fs) +_{\mathtt{ep}} n = ((f, a + n), fs)$. We use the meta variables s, s', t, and t' to range over STATE.

We model the possible runs of a program $P \in \mathtt{PROG}$ by the transition relation $\Downarrow_P \subseteq \mathtt{STATE} \times \mathtt{STATE} \times \mathbb{N}$. We write $(s, s', n) \in \Downarrow_P$ as $s \Downarrow_P^n s'$ to capture that the execution of P in state s terminates in state s' after n clock cycles. Formally, we define the relation using the derivation rules

$$\frac{s \xrightarrow{c}_P s' \quad s' \Downarrow_P^{c'} s''}{s \Downarrow_P^{c+c'} s''}\ (\mathsf{Seq}) \qquad \frac{s \xrightarrow{c}_P s' \quad \mathsf{epselect}(s') = \epsilon}{s \Downarrow_P^{c} s'}\ (\mathsf{Ter})$$

where we define the judgment $t \xrightarrow{c}_P t'$ to capture that one execution step of program P in state t takes c clock cycles and leads to state t'.

We define a small-step semantics with derivation rules for the judgment $t \xrightarrow{c}_P t'$. We make the full definition of the small-step semantics available online (as part of the addendum of this article, see Footnote 1). Below we present the rules (adc), ($\mathsf{breq\text{-}t}$) and ($\mathsf{breq\text{-}f}$) as examples:

$$\frac{\begin{array}{c} P(ep) = \mathtt{adc}\ Rd\ Rr \qquad r' = r[Rd \mapsto r(Rd) + r(Rr) + sr(C)] \\ sr' = sr[C \mapsto_s cf_1 \vee cf_2][Z \mapsto_s r'(Rd) = 0] \quad cf_1 = (r(Rd)_{[7]} \wedge r(Rr)_{[7]}) \\ cf_2 = (r(Rr)_{[7]} \wedge \neg r'(Rd)_{[7]}) \vee (\neg r'(Rd)_{[7]} \wedge r(Rd)_{[7]}) \end{array}}{(sr, m, r, st, ep) \xrightarrow{\mathtt{t}(P(ep))}_P (sr', m, r', st, ep +_{\mathtt{ep}} 1)}\ (\mathsf{adc})$$

$$\frac{P(ep) = \texttt{breq}\ epa \quad sr(Z) \neq 1}{(sr, m, r, st, ep) \xrightarrow{\texttt{t}(P(ep))}_P (sr, m, r, st, ep +_{\texttt{ep}} 1)} \ (\textsf{breq-f})$$

$$\frac{P(ep) = \texttt{breq}\ epa \quad sr(Z) = 1 \quad ep = (ep_0,\ fs) \quad ep' = (epa,\ fs)}{(sr, m, r, st, ep) \xrightarrow{\texttt{t}(P(ep))+\texttt{br}}_P (sr, m, r, st, ep')} \ (\textsf{breq-t})$$

The AVR instruction `adc` Rd Rr stores the sum of the operands and the carry flag in Rd. The instruction takes 1 clock cycle [6]. We capture the semantics of `adc` in the semantics rule (adc). We define the resulting contents of register Rd to be the sum of the original values of Rd, Rr, and C. We define the resulting status flags by sr', which maps C to 1 if there was a carry and which maps Z to 1 if the sum is zero. We define the execution point of the resulting state by $ep +_{\texttt{ep}} 1$. We capture the execution time of `adc` by the annotation $\texttt{t}(P(ep))$. Since $\texttt{t}(\texttt{adc}\ Rd\ Rr) = 1$, this annotation captures the time faithfully.

The AVR instruction `breq` epa branches on the zero flag. It takes 2 clock cycles if a jump to epa is performed (*then*-case, zero flag set) and 1 clock cycle otherwise (*else*-case) [6]. We capture the semantics of `breq` by two semantics rules. We capture the *else*-case by the rule (breq-f). We capture the condition for the *else*-case by the premise $sr(Z) \neq 1$ and the resulting execution point by $ep +_{\texttt{ep}} 1$. We capture the execution time by $\texttt{t}(P(ep))$, which is 1 by definition of `t`. We capture the semantics of the *then*-case by the rule (breq-t). We capture the condition for the *then*-case by the premise $sr(Z) = 1$ and the resulting execution point by ep', where ep' consists of the target execution point epa and the unmodified call stack. To capture the execution time, we define the constant $\texttt{br} = 1$. We define the annotation of the judgment as $\texttt{t}(P(ep)) + \texttt{br}$ to reflect the additional clock cycle that the instruction `breq` requires in the *then*-case.

Overall, the execution times of all non-branching instructions in our semantics are captured completely by the function `t`. For all branching instructions in our semantics, we add the constant `br` to the execution time `t` in the *then*-case to reflect the additional clock cycle required to jump to the *then*-branch.

Based on our operational semantics, we define the successor-relation $\rightsquigarrow_P$ such that $\texttt{ep}_1 \rightsquigarrow_P \texttt{ep}_2 \iff \exists s_1, s_2 \in \texttt{STATE} : \exists n \in \mathbb{N} : s_1 \xrightarrow{n}_P s_2 \wedge \textsf{epselect}(s_1) = \texttt{ep}_1 \wedge \textsf{epselect}(s_2) = \texttt{ep}_2$. We define the execution points that are reachable from an execution point `ep` in program P by $\textsf{reachable}_P(\texttt{ep}) := \{\texttt{ep}' \in \texttt{EPS} \mid \texttt{ep} \rightsquigarrow_P^+ \texttt{ep}'\}$.

4 Timing-Sensitive Noninterference

We capture the security requirements for AVR assembly programs based on a two-level security lattice. Its elements are security levels $\mathcal{L}$ and $\mathcal{H}$ with $\sqsubseteq := \{(\mathcal{L}, \mathcal{L}), (\mathcal{L}, \mathcal{H}), (\mathcal{H}, \mathcal{H})\}$ and least upper bound operator $\sqcup$. The security level $\mathcal{L}$ is used for attacker-visible information and $\mathcal{H}$ is used for confidential information. Each information container is annotated with a security level by a domain assignment.

Register and status-register domain assignments out of $\texttt{REG-DA} := \texttt{REG} \rightarrow \{\mathcal{L}, \mathcal{H}\}$ and $\texttt{STAT-DA} := \{C, Z\} \rightarrow \{\mathcal{L}, \mathcal{H}\}$, respectively, assign security levels to each individual register and status register. Registers $r, r' \in \texttt{REG-VAL}$ are indistinguishable with respect to $\mathtt{rda} \in \mathtt{REG\text{-}DA}$, written $r \approx_{\mathtt{rda}} r'$, if and only if $\forall x \in \mathtt{REG} : \mathtt{rda}(x) = \mathcal{L} \Rightarrow r(x) = r'(x)$, (and likewise $\approx_{\mathtt{srda}}$ for status registers).

The whole memory is annotated with a single level from $\{\mathcal{L}, \mathcal{H}\}$. For $\mathtt{md} \in \{\mathcal{L}, \mathcal{H}\}$, memories $m, m' \in \mathtt{MEM\text{-}VAL}$ are indistinguishable if $\mathtt{md} = \mathcal{L} \Rightarrow m = m'$.

The stack is annotated by a stack domain assignment out of $\mathtt{STACK\text{-}DA} := \{\mathcal{L}, \mathcal{H}\}^*$. Two stacks $l, l' \in \mathtt{STACK\text{-}VAL}$ are indistinguishable with respect to a stack domain assignment $\mathtt{sda} \in \mathtt{STACK\text{-}DA}$, written $l \simeq_{\mathtt{sda}} l'$, if and only if the stacks only differ in the contents of $\mathcal{H}$ elements until after the bottom-most $\mathcal{L}$ element. They may differ arbitrarily below the bottom-most $\mathcal{L}$ element.

Finally, states $s, s' \in \mathtt{STATE}$ are indistinguishable, written $s \approx_{\mathtt{sda},\mathtt{md},\mathtt{rda},\mathtt{srda}} s'$, if and only if their components (except the execution points) are componentwise indistinguishable. We use the meta variables $\mathtt{da}$ and $\mathtt{da}'$ to range over $\mathtt{STACK\text{-}DA} \times \{\mathcal{L}, \mathcal{H}\} \times \mathtt{REG\text{-}DA} \times \mathtt{STAT\text{-}DA}$ and write $\mathtt{da} \sqsubseteq \mathtt{da}'$ to abbreviate the straight-forward notions of partial order on all components of $\mathtt{da}$ and $\mathtt{da}'$.

We express timing-sensitive noninterference by the property TSNI.

Definition 1. *A program P satisfies TSNI starting from $\mathtt{ep}_s \in \mathtt{EPS}$ with initial and finishing domain assignments $\mathtt{da}$ and $\mathtt{da}'$ if and only if*

$$\begin{aligned}
&\forall s_0, s_0', s_1, s_1' \in \mathtt{STATE} \colon \forall n, n' \in \mathbb{N} \colon \\
&\qquad \mathsf{epselect}(s_0) = \mathtt{ep}_s \wedge \mathsf{epselect}(s_0') = \mathtt{ep}_s \wedge \\
&\qquad s_0 \approx_{\mathtt{da}} s_0' \ \wedge s_0 \Downarrow_P^n s_1 \wedge \ s_0' \Downarrow_P^{n'} s_1' \\
&\qquad\quad \Rightarrow s_1 \approx_{\mathtt{da}'} s_1' \wedge n = n'
\end{aligned}$$

The initial and finishing domain assignments should be chosen to reflect which inputs and outputs are visible to an attacker. If a program then satisfies TSNI, an attacker cannot distinguish between two secret inputs to the program by observing the program's output or execution time. That is, TSNI guarantees secure information flow and the absence of timing-side-channel vulnerabilities.

5 Timing-Sensitive Type System for AVR Assembly

We provide a security type system for checking AVR assembly programs against timing-side-channel vulnerabilities. We define the type system such that programs are only typable if their execution time does not depend on secret information. Furthermore, our definition of the type system rules out undesired direct and indirect information flow in typable programs.

5.1 Precomputation of Control-Dependence Regions

To check whether the control flow of a program influences attacker-observable information or the running time, the control flow must be known. Since AVR assembly is an unstructured language, the control dependencies of a program are not structurally encoded in its syntax. To address this, we approximate the control-dependence regions in a program using Safe Over Approximation Properties (SOAPs). To be able to define typing rules that compare the execution

SOAP1 $\forall \mathsf{ep}_1, \mathsf{ep}_2, \mathsf{ep}_3 \in \mathsf{EPS}$ such that $\mathsf{ep}_1 \rightsquigarrow_P \mathsf{ep}_2$, $\mathsf{ep}_1 \rightsquigarrow_P \mathsf{ep}_3$ and $\mathsf{ep}_2 \neq \mathsf{ep}_3$ exactly one of the following holds
- $\mathsf{ep}_2 \in \mathsf{region}^i_P(\mathsf{ep}_1)$ and $\mathsf{ep}_3 \in \mathsf{region}^j_P(\mathsf{ep}_1)$ for unique $i, j \in \{1, 2\}, i \neq j$
- $\mathsf{ep}_i \in \mathsf{region}^1_P(\mathsf{ep}_1)$ and $\mathsf{jun}_P(\mathsf{ep}_1) = \mathsf{ep}_j$ for unique $i, j \in \{2, 3\}, i \neq j$

SOAP2 $\forall \mathsf{ep} \in \mathsf{EPS} : \mathsf{region}^1_P(\mathsf{ep}) \cap \mathsf{region}^2_P(\mathsf{ep}) = \emptyset$.

SOAP3 $\forall \mathsf{ep}_1, \mathsf{ep}_2, \mathsf{ep}_3 \in \mathsf{EPS}$ and $\forall i \in \{1, 2\}$, if $\mathsf{ep}_2 \in \mathsf{region}^i_P(\mathsf{ep}_1)$ and $\mathsf{ep}_2 \rightsquigarrow_P \mathsf{ep}_3$, then either $\mathsf{ep}_3 \in \mathsf{region}^i_P(\mathsf{ep}_1)$ or $\mathsf{jun}_P(\mathsf{ep}_1) = \mathsf{ep}_3$.

SOAP4 $\forall \mathsf{ep}_1, \mathsf{ep}_2 \in \mathsf{EPS}$ and $\forall i \in \{1, 2\}$, if $\mathsf{ep}_2 \in \mathsf{region}^i_P(\mathsf{ep}_1)$ and $\neg\exists \mathsf{ep}_3 \in \mathsf{EPS} : \mathsf{ep}_2 \rightsquigarrow_P \mathsf{ep}_3$, then $\mathsf{jun}_P(\mathsf{ep}_1)$ is undefined.

Figure 1. Safe Overapproximation Properties

time of *then*- and *else*-branches, we distinguish between two control-dependence regions for each branching.

Formally, we define the functions $\mathsf{region}^1_P, \mathsf{region}^2_P : \mathsf{EPS} \to \mathcal{P}(\mathsf{EPS})$ and $\mathsf{jun}_P : \mathsf{EPS} \rightharpoonup \mathsf{EPS}$ to be a safe over approximation of program P's control-dependence regions if they satisfy the SOAPs in Figure 1. That is, if the branches of each branching instruction are captured by the two regions of the instruction, if the regions of each instruction are disjoint, if a step in a region either leads to the junction point or another point in the region, and if all regions that contain an instruction without a successor have no junction point. In the following we only consider functions region^{then}_P and region^{else}_P that satisfy the SOAPs.

We define $\mathsf{region}_P(\mathsf{ep}) := \mathsf{region}^1_P(\mathsf{ep}) \cup \mathsf{region}^2_P(\mathsf{ep})$. For a branching instruction at execution point ep we denote the region from $\{\mathsf{region}^1_P, \mathsf{region}^2_P\}$ that contains the branch target by $\mathsf{region}^{then}_P(\mathsf{ep})$ and the other region by $\mathsf{region}^{else}_P(\mathsf{ep})$.

To distinguish loops from branchings, we define the predicate $\mathsf{loop}_P(\mathsf{ep}) := \exists \mathsf{ep}' \in \mathsf{region}_P(\mathsf{ep}) : \mathsf{ep} \rightsquigarrow^+_P \mathsf{ep}'$ *contains a back edge*, which captures whether an execution point is the header of a natural loop. We assume that programs contain only natural loops.

5.2 Typing Rules

Given a program P with control-dependence regions region^{then}_P and region^{else}_P, we define the typability of P with respect to an initial domain assignment, a finishing domain assignment, and a security environment. We define a security environment to be a function $se : \mathsf{EPS} \to \{\mathcal{L}, \mathcal{H}\}$ that assigns a security level to every execution point in the program. Moreover, we define the type system such that se maps all execution points to $\mathcal{H}$ whose execution depends on secret information. Finally, we define a program to be typable if domain assignments for all intermediate states in the program execution exist such that, for each execution point ep_i, a judgment of the form

$$P, \mathsf{region}^{then}_P, \mathsf{region}^{else}_P, se, \mathsf{ep}_i : \\ (\mathtt{sda}_{\mathsf{ep}_i}, \mathtt{md}_{\mathsf{ep}_i}, \mathtt{rda}_{\mathsf{ep}_i}, \mathtt{srda}_{\mathsf{ep}_i}) \vdash (\mathtt{sda}'_{\mathsf{ep}_j}, \mathtt{md}'_{\mathsf{ep}_j}, \mathtt{rda}'_{\mathsf{ep}_j}, \mathtt{srda}'_{\mathsf{ep}_j})$$

is derivable that relates the domain assignments of ep_i to domain assignments that are at most as restrictive as the domain assignments of all successors of ep_i.

$$\frac{\begin{array}{c} P(ep) = \mathtt{adc}\ Rd\ Rr \\ erg = \mathtt{rda}(Rd) \sqcup \mathtt{rda}(Rr) \sqcup se(ep) \sqcup \mathtt{srda}(C) \\ \mathtt{rda}' = \mathtt{rda}[Rd \mapsto erg] \qquad \mathtt{srda}' = \mathtt{srda}[C \mapsto erg][Z \mapsto erg] \end{array}}{P, \cdots, ep : (\mathtt{sda}, \mathtt{md}, \mathtt{rda}, \mathtt{srda}) \vdash (\mathtt{sda}, \mathtt{md}, \mathtt{rda}', \mathtt{srda}')} \ (\textsf{t-adc})$$

$$\frac{\begin{array}{c} \exists instr \in \{\mathtt{breq},\ \mathtt{brne}\} : P(ep) = instr\ epa \\ se(ep) \sqcup \mathtt{srda}(Z) = \mathcal{L} \end{array}}{P, \cdots, ep : (\mathtt{sda}, \mathtt{md}, \mathtt{rda}, \mathtt{srda}) \vdash (\mathtt{sda}, \mathtt{md}, \mathtt{rda}, \mathtt{srda})} \ (\textsf{t-brZ-l})$$

$$\frac{\begin{array}{c} \exists instr \in \{\mathtt{breq},\ \mathtt{brne}\} : P(ep) = instr\ epa \\ \neg \mathsf{loop}_P(ep) \qquad se(ep) \sqcup \mathtt{srda}(Z) = \mathcal{H} \qquad se(ep) = \mathcal{H} \\ \forall ep' \in \mathsf{region}_P(ep) : se(ep') = \mathcal{H} \qquad \mathtt{sda}' = \mathsf{lift}(\mathtt{sda}, \mathcal{H}) \\ \mathsf{branchtime}_P^{then}(ep) + \mathtt{br} = \mathsf{branchtime}_P^{else}(ep) \end{array}}{P, \cdots, ep : (\mathtt{sda}, \mathtt{md}, \mathtt{rda}, \mathtt{srda}) \vdash (\mathtt{sda}', \mathtt{md}, \mathtt{rda}, \mathtt{srda})} \ (\textsf{t-brZ-h})$$

Figure 2. Selected typing rules

Definition 2. *A program P with control-dependence regions region_P^{then} and region_P^{else} is typable with starting execution point ep_s, initial domain assignments $\mathtt{da}_{\mathsf{ep}_s}$, finishing domain assignments $\mathtt{da}_f$, and security environment se, written*

$$P, \mathsf{region}_P^{then}, \mathsf{region}_P^{else}, se, \mathsf{ep}_s : \mathtt{da}_{\mathsf{ep}_s} \Vdash \mathtt{da}_f,$$

if and only if for every $\mathsf{ep} \in \mathsf{reachable}_P(\mathsf{ep}_s)$ there exist domain assignments $\mathtt{da}_{\mathsf{ep}}$ such that for all $\mathsf{ep}_i, \mathsf{ep}_j \in \mathsf{reachable}_P(\mathsf{ep}_s) \cup \{\mathsf{ep}_s\}$, both,

1. *if $\mathsf{ep}_i \rightsquigarrow_P \mathsf{ep}_j$ then $\exists \mathtt{da}'_{\mathsf{ep}_j} : \mathtt{da}'_{\mathsf{ep}_j} \sqsubseteq \mathtt{da}_{\mathsf{ep}_j} \wedge P, \cdots, \mathsf{ep}_i : \mathtt{da}_{\mathsf{ep}_i} \vdash \mathtt{da}'_{\mathsf{ep}_j}$.*
2. *if there exists no $\mathsf{ep}_k \in \mathsf{reachable}_P(\mathsf{ep}_s)$ such that $\mathsf{ep}_i \rightsquigarrow_P \mathsf{ep}_k$ then $\mathtt{da}_{\mathsf{ep}_i} \sqsubseteq \mathtt{da}_f$ and $P, \cdots, \mathsf{ep}_i : \mathtt{da}_{\mathsf{ep}_i} \vdash \mathtt{da}_{\mathsf{ep}_i}$ is derivable.*

Note that our definition of typability imposes constraints on domain assignments of consecutive execution points (see Condition 1 in Definition 2) as well as on domain assignments upon termination (see Condition 2 in Definition 2).

We define the derivability of the typing judgment $P, \cdots, \mathsf{ep}_i : \mathtt{da}_{\mathsf{ep}_i} \vdash \mathtt{da}'_{\mathsf{ep}_j}$ by typing rules for the individual AVR instructions. In this section we present the rules (t-adc), (t-brZ-l), and (t-brZ-h), defined in Figure 2. We make the full definition of the type system available online (see Footnote 1).

We define the derivable typing judgments for execution points that point to `adc` instructions by the typing rule (t-adc). In this typing rule, we raise the security levels of the registers and status flags modified by `adc` to the least upper bound of the security levels of the summands, the carry flag and the security environment. By raising the security levels, we ensure the absence of flows from $\mathcal{H}$ summands, carry, or branching conditions to an $\mathcal{L}$ sum, carry, or zero flag.

We define the derivable typing judgments for the instructions `breq` and `brne`, which jump conditionally on the zero flag, by two typing rules. By the typing rule (t-brZ-l) we define the derivable judgments for jumps that only depend on $\mathcal{L}$ information. We capture the condition that the jump only depends on $\mathcal{L}$ information by a premise that requires the security environment and the zero flag to have the security level $\mathcal{L}$. That is, the execution of the conditional jump instruction and the condition for jumping are required to only depend on $\mathcal{L}$ information.

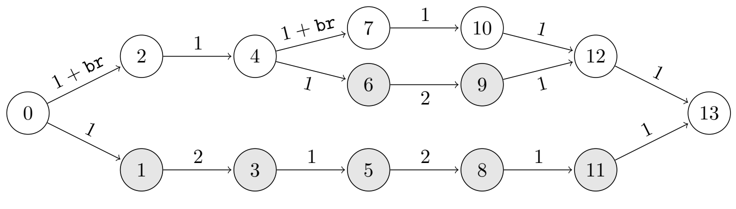
We define the derivable judgments such that they do not modify any security levels, because a conditional jump instruction does not modify any information. By the typing rule $(\mathsf{t\text{-}brZ\text{-}h})$, we define the derivable judgments for jumps that depend on $\mathcal{H}$ information. We forbid loops depending on $\mathcal{H}$ information to avoid leakage to the number of iterations. We allow branchings on $\mathcal{H}$ information under the following conditions. The security environment must reflect the dependence of the branches on $\mathcal{H}$ information. The security levels of the stack must reflect that the height of the stack could differ across the branches (expressed using the function lift that lifts all elements of $\mathtt{sda}$ to $\mathcal{H}$ recursively). Finally, the execution time required for the *else*-branch must be equal to the time for jumping to and executing the *then*-branch. We capture the time required for the jump by $\mathtt{br}$. We capture the time required to execute a branch by the function $\mathsf{branchtime}_P^r$, where $r \in \{then, else\}$.

Definition 3. *The function* $\mathsf{branchtime}_P^r$ *is defined recursively as*

$$\mathsf{branchtime}_P^r(\mathsf{ep}) := \sum_{\substack{\mathsf{ep}_i \in \mathsf{region}_P^r(\mathsf{ep}) \\ \mathsf{ep}_i \neq \mathsf{ep}}} \Big(\mathsf{t}(P(\mathsf{ep}_i)) - \mathsf{branchtime}_P^{then}(\mathsf{ep}_i)\Big)$$

We define the function $\mathsf{branchtime}^r(\mathsf{ep}_0)$ of a non-nested branching ep_0, such that it sums up the execution time of all instructions inside the branching. A recursion is not required, as for all $\mathsf{ep}' \in \mathsf{region}^r(\mathsf{ep}_0)$ it holds that $\mathsf{region}^{then}(\mathsf{ep}') = \emptyset$. Now assume ep_1 and ep_2 are branching instructions with $\mathsf{ep}_2 \in \mathsf{region}^r(\mathsf{ep}_1)$. Then only one branch of ep_2 is executed, but the positive part of $\mathsf{branchtime}^r(\mathsf{ep}_1)$ sums up the execution time of both branches of ep_2. We take care of this by subtracting the execution time of the *then*-branch. By typability, it is ensured that both branches of ep_2 execute in the same time, making the execution time of ep_1 independent of the branch taken at ep_2.

Example 1. The following control flow graph is annotated with execution times.



The *then*-branches are white, the *else*-branches are gray. Consider the paths from Node 4 to 12. They don't contain nested branches. We get $\mathsf{branchtime}_P^{then}(4) = 2$ and $\mathsf{branchtime}_P^{else}(4) = 3$. For the paths from Node 0 to Node 13, there is one nested branching, namely the previously considered branching at Node 4. We get

$$\begin{aligned}\mathsf{branchtime}_P^{then}(0) &= 1+1+1+2+1+1+1 - \sum_{\mathsf{ep}_i \in \mathsf{region}_P^{then}(0)} \mathsf{branchtime}_P^{then}(\mathsf{ep}_i) \\ &= 1+1+1+2+1+1+1 - \mathsf{branchtime}_P^{then}(4) = 6\end{aligned}$$

Only $\mathsf{branchtime}_P^{then}(4)$ is subtracted because all other points in the region have 0 branchtime. $1 + \mathtt{br}$ is counted as 1 because $\mathtt{br}$ is handled in the typing rule. $\diamondsuit$

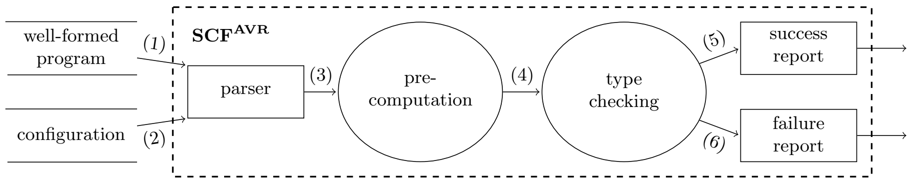


Figure 3. Data flow diagram of the analysis process in SCF

5.3 Soundness

We ensure that our security type system provides reliable security guarantees about AVR programs. To this end, we prove the following soundness theorem.

Theorem 1 (Soundness). *If* $P, \mathsf{region}_P^{then}, \mathsf{region}_P^{else}, se, \mathtt{ep}_s : \mathtt{da}_{\mathtt{ep}_s} \Vdash \mathtt{da}_f$, *then* P *satisfies TSNI starting from* $\mathtt{ep}_s$ *with the initial and finishing domain assignments* $\mathtt{da}_{\mathtt{ep}_s}$ *and* $\mathtt{da}_f$*.*

Proof Sketch. We apply an unwinding technique and prove local respect and step consistency for each typable AVR assembly instruction in our semantics. To prove that no secret information interferes with the execution time, we formulate and prove a lemma stating that secret-dependent branches are constant-time. □

Theorem 1 states that the type system is sound with respect to the property TSNI. That is, all typable programs are free of timing-side-channel vulnerabilities with respect to TSNI. We make the full proof available online (as part of the addendum of this article, see Footnote 1).

Proving the soundness of a security type system with respect to a security property is an established technique used, e.g., in [2,9,33,51]. In general, timing-side-channel vulnerabilities might occur in practice despite soundness proofs [40]. This criticism does not apply to our approach because our semantics is based on the explicit specification of execution times in [6].

6 Automatically Analyzing AVR Assembly Programs

We create the Side-Channel Finder$^{\mathrm{AVR}}$ (SCF$^{\mathrm{AVR}}$) to automatically analyze AVR programs with respect to timing-side-channel vulnerabilities. From now we omit the superscript of SCF$^{\mathrm{AVR}}$. We make the tool available online (see Footnote 1).

To demonstrate the capabilities of SCF, we apply it to a self-implemented primitive and to off-the-shelf implementations from the crypto library μNaCl.

6.1 The Side-Channel Finder$^{\mathrm{AVR}}$

Our analysis of AVR assembly programs consists of three steps that are illustrated in Figure 3. The dashed box represents the parts of the analysis that we automate in SCF. The first step is to parse the analysis inputs. We convert the inputs, namely an AVR program (1) and a configuration file (2), to an internal representation. The configuration file specifies a starting execution point and initial and finishing domain assignments. The second step is to precompute (3) the control-dependence regions of the AVR assembly program. The third step is the timing-sensitive information flow analysis (4) of the program. If the analysis is successful, we report the success (5). Otherwise, we return a failure report (6).

Implementation. The tool SCF is our implementation of this three-step analysis procedure in roughly 1,250 lines of Python code. SCF takes as the first input an object dump file of the program to analyze. The object dump file can be generated with the AVR compiler toolchain and contains the full program in assembly form. We implement a simple regex-based parser to transform an object dump file into a program representation according to our syntax in Section 3.1. As the second input, SCF takes the analysis configuration in JSON format. Our parser infers the registers of function arguments from high-level code according to the AVR calling conventions [23] and the given configuration file.

We implement the precomputation according to the SOAPs for control-dependence regions from Section 5.1. Our implementations is based on a method from [25] and uses the graph library NetworkX [27] to compute dominators.

To realize the information-flow analysis in the third step, we implement our type system from Section 5.2. We represent each instruction as a class that contains the corresponding typing rule and the corresponding execution time according to our definition of `t` for ATmega processors in Table 1. We implement type checking as a fixed-point iteration.

If there is no error detected during type checking, we report the result `SUCCESS`. Otherwise, we report a failure. We provide an error message that specifies the origin of the failure. The concrete error messages are:

- `LOOP_ON_SECRET_DATA`, if there is a loop in a high security environment,
- `TIMING_LEAK`, if there is a violation of a branchtime condition,
- `INFORMATION_LEAK`, if the inferred domain assignments are more restrictive than allowed by the given configuration.

6.2 Timing-Side-Channel Analysis of μNaCl

We demonstrate how to analyze real-world cryptographic implementations with SCF at the example of μNaCl. μNaCl [29] is specifically made for AVR microcontrollers and was developed with a focus on providing constant-time implementations of cryptographic primitives. We analyze the constant-time string-comparison primitive from μNaCl and an alternative implementation of string comparison that is vulnerable to timing-side-channel attacks. We also analyze the μNaCl default stream cipher Salsa20 and its variant XSalsa20, and the μNaCl default Message-Authentication Code Poly1305. We expected these implementations to be secure because side channels were a focus in the development of μNaCl [29]. Our analysis with SCF confirms that these implementations are secure with respect to the timing-sensitive property TSNI. The analysis is fully automatic and does not require any source code modifications[2] to μNaCl.

[2] All crypto functions in μNaCl satisfy the assumption of a unique return instruction.

String Comparison. Consider the following two implementations of string comparison where `n` is the length of the strings to be compared.

```
for(i = n; i != 0; i--)
  if(x[i-1] != y[i-1])
    break;
return i;
```

```
crypto_uint16 d = 0;
for(i = 0; i < n; i++)
  d |= x[i]^y[i];
return (1&((d-1)>>8))-1;
```

The first implementation aborts the comparison at the first mismatch. The second implementation always iterates over the entire string. If the implementations are used, e.g., to verify passwords, the first implementation leaks the amount of correct characters in the password via a timing channel, while the second implementation is constant-time.

Using SCF, one can check for such vulnerabilities automatically. We analyzed the implementations for $n = 16$. Since either of the source-level inputs could be the actual password, we run SCF with the security level $\mathcal{H}$ for both inputs. In the parsing phase, this domain assignment is translated according to the calling conventions, so that registers r_{22} to r_{25} are initially $\mathcal{H}$. To check for timing side channels, we assume that the attacker cannot observe the output directly but only the timing. Hence, we also set the security level of the result to $\mathcal{H}$. On the first program, SCF detects a vulnerability. The output of SCF looks as follows.

```
"result_code":3,
"execution_point":{
  "address":"0x1a", "function":"verify_leaky_16"},
"result":"LOOP_ON_SECRET_DATA"
```

SCF points to the address at which the vulnerability was detected and also hints at the reason, namely a loop on secret data. The address "0x1a" points to the if-statement that leads to early abortion of the string comparison.

On the second implementation of string comparison, SCF reports a successful analysis. The implementation is typable. By Theorem 1, the implementation is secure against timing-side-channel vulnerabilities with respect to TSNI.

The second implementation of string comparison is used in μNaCl. We successfully analyzed the μNaCl string comparison functions `crypto_verify16` and `crypto_verify32` that both use the second implementation. Both functions are secure with respect to TSNI.

Salsa20 and Poly1305. SCF is also able to analyze more complex cryptographic implementations than a password verification. We apply SCF to the implementations of Salsa20, XSalsa20, and Poly1305 in the library μNaCl.

The cipher Salsa20 [13] is part of the eSTREAM portfolio of stream ciphers. The specification of Salsa20 avoids S-box lookups and integer multiplications as sources of potential timing vulnerabilities. We analyze the μNaCl implementations of Salsa20 and XSalsa20 (a variant with a longer nonce [14]). The parameters of both, the Salsa20 and XSalsa20 implementations, are the secret key `k`, a nonce `n`, the location for the cipher output `c`, and the message length `clen`.

We consider the key `k` and the nonce `n` secret and assign security level $\mathcal{H}$. Furthermore, we consider an attacker who can only observe the timing of an execution, and we assign the level $\mathcal{H}$ to the cipher output stored in `c` and to the return value (status) of the functions. We consider the message length `clen` visible to the attacker and assign level $\mathcal{L}$. The analysis of Salsa20 and XSalsa20 with SCF is successful, i.e., the functions are secure with respect to TSNI.

Poly1305 [15] is a MAC (Message-Authentication Code) based on secret-key encryption. While the original definition of Poly1305 is based on AES, the implementation in μNaCl is based on Salsa20. The parameters of the Poly1305 implementation in μNaCl are the secret key `k`, the message `in`, the message length `inlen`, and the location for the resulting authenticator `out`.

We analyze the μNaCl implementation of Poly1305 with SCF. Again we consider only the message length `inlen` visible to the attacker. SCF reports a successful analysis. The function is typable and hence satisfies TSNI.

Analysis Setup. From version 20140813 of μNaCl we analyzed `crypto_verify16`, `crypto_verify32`, `crypto_stream_salsa20`, `crypto_stream_xsalsa20`, as well as `crypto_onetimeauth_poly1305`. We obtained the object dump using `avr-gcc` in version 4.8.1 and `avr-objdump`. We removed the flag `--mcall-prologues` from the μNaCl makefile to obtain the full assembly code.

7 Related Work

Timing Side Channels. Already in 1996, Kocher [31] described how to extract a secret key from a cryptosystem by measuring the running time. Brumley and Boneh [17] showed that timing attacks can be carried out remotely, which makes them particularly dangerous. In general, timing vulnerabilities can be due to different factors, e.g., secret-dependent branches with different execution times [31], branch prediction units [1], or caches [12]. In this article, we consider a platform without optimizations like branch prediction units and caches.

Timing vulnerabilities can be avoided by design as, e.g., in μNaCl [29] or transformed out of existing implementations [2, 11, 33, 41]. The use of program transformations does not always lead to implementations without timing-side-channel vulnerabilities in practice [40]. For the secure design of selected implementations from μNaCl, we certify timing-sensitive noninterference based on the official specification of execution times in [6].

Side-Channels on AVR Microcontrollers. Hardware cryptographic engines on AVR microcontrollers have been successfully attacked through side channels by Kizhvatov [30], O'Flynn and Chen [43], and Ronen et al. [47].

An alternative to hardware-accelerated cryptography are cryptographic implementations in software, e.g., in cryptographic libraries like μNaCl [29]. For an informed use of software implementations, reliable security guarantees are desirable. Our tool SCF can check AVR assembly programs and provide such guarantees. It complements existing techniques like the ChipWhisperer toolbox [42] that supports mounting side-channel attacks on AVR microcontrollers.

Timing-Sensitive Information Flow Analysis. Timing-sensitive security type systems were developed for an imperative programming language and a while language already by Volpano and Smith [49] in 1997 and by Agat [2] in 2000. Agat's type system was extended to a JavaCard-like bytecode language by Hedin and Sands [28]. For an intermediate language in the CompCert verified C compiler, timing-sensitive information flow was considered by Barthe et al. [9]. Agat [2] and Köpf and Mantel [33] propose type systems that transform programs to remove timing-side-channel vulnerabilities. Our type system for AVR assembly is not transforming. However, the AVR instruction set contains a `nop` command that could be used to realize a transforming type system.

Recently, Zhang, Askarov, and Myers [50] proposed a timing-sensitive type system that takes into account a contract for the interaction of programs with the hardware design. To check whether hardware adheres to such a contract, Zhang, Wang, Suh, and Myers [51] introduce a hardware design language with type annotations and a corresponding timing-sensitive security type system.

Existing tools for timing-sensitive program analysis include Side Channel Finder [38] for Java, which checks for secret-dependent loops and branchings using a type system, and CacheAudit [22] for x86 binaries, which quantifies the leakage through cache-based timing channels using abstract interpretation.

To our knowledge, we propose the first information flow analysis and analysis tool for checking AVR assembly programs against timing side channels.

8 Conclusion

In this article, we have shown how an analysis framework for timing side channels in real-world crypto implementations can be realized. We proposed a security type system, a timing-sensitive operational semantics, a soundness result for our type system, and our tool SCF for automatically verifying the absence of information leaks (including timing side channels) in AVR programs. We exploited the predictability of execution times on 8-bit AVR processors and showed how AVR can be used as a platform for language-based approaches to timing-sensitive information flow analysis. SCF is an academic prototype, but - as we have shown - it is suitable for verifying real-world crypto implementations from μNaCl.

Based on this initial step, we plan to increase the coverage of our framework from currently 36% of the 8-bit AVR instruction set to the entire 8-bit AVR instruction set. We plan to grow SCF so that it can be broadly applied to off-the-shelf AVR assembly programs. With the extended SCF, the verification of entire crypto libraries will be an interesting direction. Another interesting direction would be to consider attackers who exploit hardware features (e.g., interrupts).

Acknowledgements. We thank the anonymous reviewers for their constructive comments. We also thank Ximeng Li, Johannes Schickel, and Artem Starostin for helpful discussions. This work has been funded by the DFG as part of Project E3 “Secure Refinement of Cryptographic Algorithms” within the CRC 1119 CROSSING.

References

1. Aciiçmez, O., Koç, Ç.K., Seifert, J.P.: Predicting secret keys via branch prediction. In: CT-RSA. pp. 225–242 (2007)

2. Agat, J.: Transforming out timing leaks. In: POPL. pp. 40–53 (2000)
3. Appel, A.W.: Modern Compiler Implementation in Java. Cambridge University Press (2002)
4. Atmel Corporation: Atmel ATmega2564RFR2 / ATmega1284RFR2 / ATmega644RFR2 Datasheet. Rev. 42073B-MCU Wireless-09/14 (2014)
5. Atmel Corporation: Atmel ATmega640 / V-1280 / V-1281 / V-2560 / V-2561 / V Datasheet. Rev. 2549Q–AVR–02/2014 (2014)
6. Atmel Corporation: Atmel AVR 8-bit Instruction Set: Instruction Set Manual. Rev. 0856K–AVR–05/2016 (2016)
7. Atmel Corporation: Automotive AVR Microcontrollers (2016), `http://www.atmel.com/products/microcontrollers/avr/Automotive_AVR.aspx`, Accessed 21.03.17
8. Atmel Corporation: Rad Tolerant Devices (2016), `http://www.atmel.com/products/rad-hard/rad-tolerant-devices/`, Accessed 21.03.17
9. Barthe, G., Betarte, G., Campo, J.D., Luna, C., Pichardie, D.: System-level Non-interference for Constant-time Cryptography. In: CCS. pp. 1267–1279 (2014)
10. Barthe, G., Pichardie, D., Rezk, T.: A Certified Lightweight Non-Interference Java Bytecode Verifier. In: ESOP. pp. 125–140 (2007)
11. Barthe, G., Rezk, T., Warnier, M.: Preventing timing leaks through transactional branching instructions. ENTCS 153(2), 33–55 (2006)
12. Bernstein, D.J.: Cache-timing attacks on AES. Tech. rep., University of Illinois at Chicago (2005)
13. Bernstein, D.J.: The Salsa20 Family of Stream Ciphers. In: New Stream Cipher Designs. pp. 84–97 (2008)
14. Bernstein, D.J.: Extending the Salsa20 nonce. In: SKEW (2011)
15. Bernstein, D.J.: The Poly1305-AES Message-Authentication Code. In: FSE. pp. 32–49 (2011)
16. Brumley, B.B., Tuveri, N.: Remote Timing Attacks Are Still Practical. In: ESORICS. pp. 355–371 (2011)
17. Brumley, D., Boneh, D.: Remote Timing Attacks Are Practical. Computer Networks 48(5), 701–716 (2005)
18. Cohen, E.S.: Information Transmission in Sequential Programs. Foundations of Secure Computation pp. 297–335 (1978), Academic Press
19. Das Labor: AVR-Crypto-Lib (2014), `http://avrcryptolib.das-labor.org/trac`, Accessed 23.03.2017
20. Denning, D.E.: A lattice model of secure information flow. Communications of the ACM 19(5), 236–243 (1976)
21. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Communications of the ACM 20(7), 504–513 (1977)
22. Doychev, G., Köpf, B., Mauborgne, L., Reineke, J.: Cacheaudit: A tool for the static analysis of cache side channels. ACM TISSEC 18(1), 4:1–4:32 (2015)
23. Editors of the GCC Wiki: GCC Wiki page on avr-gcc: Calling Convention (2016), `https://gcc.gnu.org/wiki/avr-gcc#Calling_Convention`, Accessed 15.04.17
24. Fardan, N.J.A., Paterson, K.G.: Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In: S&P. pp. 526–540 (2013)
25. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM TOPLAS 9(3), 319–349 (1987)
26. Goguen, J.A., Meseguer, J.: Security policies and security models. In: S&P. pp. 11–20 (1982)
27. Hagberg, A.A., Schult, D.S., Swart, P.J.: Exploring Network Structure, Dynamics, and Function using NetworkX. In: SciPy. pp. 11–15 (2008)

28. Hedin, D., Sands, D.: Timing aware information flow security for a javacard-like bytecode. ENTCS 141(1), 163–182 (2005)
29. Hutter, M., Schwabe, P.: NaCl on 8-bit AVR Microcontrollers. In: AFRICACRYPT. pp. 156–172 (2013)
30. Kizhvatov, I.: Side Channel Analysis of AVR XMEGA Crypto Engine. In: WESS. pp. 8:1–8:7 (2009)
31. Kocher, P.C.: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: CRYPTO. pp. 104–113 (1996)
32. Kocher, P.C., Jaffe, J., Jun, B.: Differential Power Analysis. In: CRYPTO. pp. 388–397 (1999)
33. Köpf, B., Mantel, H.: Transformational typing and unification for automatically correcting insecure programs. Int. J. of Inf. Sec. 6(2), 107–131 (2007)
34. Kucuk, G., Basaran, C.: Reducing energy dissipation of wireless sensor processors using silent-store-filtering motecache. In: PATMOS. pp. 256–266 (2006)
35. Liu, A., Ning, P.: TinyECC: A Configurable Library for Elliptic Curve Cryptography in Wireless Sensor Networks. In: IPSN. pp. 245–256 (2008)
36. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-Level Cache Side-Channel Attacks are Practical. In: S&P. pp. 605–622 (2015)
37. Lortz, S., Mantel, H., Starostin, A., T.Bähr, Schneider, D., Weber, A.: Cassandra: Towards a Certifying App Store for Android. In: SPSM. pp. 93–104 (2014)
38. Lux, A., Starostin, A.: A tool for static detection of timing channels in java. J. of Crypt. Eng. 1(4), 303–313 (2011)
39. Mantel, H.: Information Flow and Noninterference. In: Encyclopedia of Cryptography and Security, pp. 605–607. Springer, 2nd edn. (2011)
40. Mantel, H., Starostin, A.: Transforming out timing leaks, more or less. In: ESORICS. pp. 447–467 (2015)
41. Molnar, D., Piotrowski, M., Schultz, D., Wagner, D.: The program counter security model: Automatic detection and removal of control-flow side channel attacks. In: ICISC. pp. 156–168 (2005)
42. O'Flynn, C., Chen, Z.: Chipwhisperer: An open-source platform for hardware embedded security research. In: COSADE. pp. 243–260 (2014)
43. O'Flynn, C., Chen, Z.: Power analysis attacks against ieee 802.15.4 nodes. In: COSADE. pp. 55–70 (2016)
44. Page, D.: Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. IACR Cryptology ePrint Archive 2002(169), 1–23 (2002)
45. Pastrana, S., Tapiador, J., Suarez-Tangil, G., Peris-López, P.: AVRAND: A Software-Based Defense Against Code Reuse Attacks for AVR Embedded Devices. In: DIMVA. pp. 58–77 (2016)
46. Prosser, R.T.: Applications of boolean matrices to the analysis of flow diagrams. In: EJCC. pp. 133–138 (1959)
47. Ronen, E., O'Flynn, C., Shamir, A., Weingarten, A.O.: IoT Goes Nuclear: Creating a ZigBee Chain Reaction. In: S&P. pp. 195–212 (2017)
48. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal On Selected Areas In Communications 21(1), 5–19 (2003)
49. Volpano, D., Smith, G.: Eliminating Covert Flows with Minimum Typings. In: CSFW. pp. 156–168 (1997)
50. Zhang, D., Askarov, A., Myers, A.C.: Language-Based Control and Mitigation of Timing Channels. In: PLDI. pp. 99–109 (2012)
51. Zhang, D., Wang, Y., Suh, G.E., Myers, A.C.: A Hardware Design Language for Timing-Sensitive Information-Flow Security. In: ASPLOS. pp. 503–516 (2015)