GARUDA 2.0

Sage Sefton
ss557415@ohio.edu
Ohio University

Avinash Karanth
karanth@ohio.edu
Ohio University

2020
Fall

EXE Input Fields	f_{Ei}	::=	$f_{Ei_1} \mid \cdots \mid f_{Ei_k}$
EXE Output Fields	f_{Eo}	::=	$f_{Eo_1} \mid \cdots \mid f_{Eo_k}$
MEM Input Fields	f_{Mi}	::=	$f_{Mi_1} \mid \cdots \mid f_{Mi_k}$
Fields	f	::=	$f_{Ei} \mid f_{Eo} \mid f_{Mi}$
Inputs	i	::=	$\{f_{i_1} = v_{i_1},\ \ldots\ ,\ f_{i_k} = v_{i_k}\}$
Outputs	o	::=	$\{f_{o_1} = v_{o_1},\ \ldots\ ,\ f_{o_k} = v_{o_k}\}$
Obfuscation Fxn	Φ	::=	$f \to f'$

Figure 1: Definitions in GARUDA 2.0.

1 Background

GARUDA was a system that operated on a two stream model of the "user" or program, and that which executed the instructions. GARUDA 2.0, we extend, or refine based on the point of view, this model to operate between the Execution (EXE) and Memory (MEM) stages of the pipeline. We assume a monitor theory shown by the graphic Fig **??**. The monitor appears to operate on a single stream, but the two hanging connections can be considered the two streams of the former model.

The most obvious change we need to make is to enable encryption and decryption of the EffAddr. Our attack vector focuses on a trojan targeting the state register between the EXE and MEM stages. They could cache these addresses and try to access that memory later, or they may try to use them as a side channel. Either way, allowing an adversary to track our memory accesses is clearly problematic. However, when we refer to "encryption", we really mean some reversible obfuscation function. We employ the proof assistant, Coq, to prove the EN and DE blocks are exact inverses. GARUDA is written in Coq for exactly this reason, so requiring a proof that $\forall P, DE(ENP) = P$ should be sufficient.

$$
\begin{array}{rlll}
\text{Predicates} & a, b & ::= & 0 \quad \textit{False} \\
& & \mid & 1 \quad \textit{Identity} \\
& & \mid & f = n \quad \textit{Test} \\
& & \mid & a + b \quad \textit{Sum} \\
& & \mid & a \cdot b \quad \textit{Product} \\
& & \mid & \neg\, a \quad \textit{Negation}
\end{array}
$$

Figure 2: Predicates in GARUDA 2.0 act as a boolean algebra.

$$
\begin{array}{rlll}
\text{Policies} & p, q & ::= & \mathbf{test}(a) \quad \textit{Test} \\
& & \mid & (\Phi_{Encrypt}, \Phi_{Decrypt}) \quad \textit{Obfuscate} \\
& & \mid & inj_{Eo} \quad \textit{Injection Execution Output} \\
& & \mid & inj_{Mi} \quad \textit{Injection Memory Input} \\
& & \mid & f \leftarrow n \quad \textit{Update} \\
& & \mid & p + q \quad \textit{Choice} \\
& & \mid & p \cdot q \quad \textit{Sequential Concatenation}
\end{array}
$$

Figure 3: Syntax of Policies in GARUDA 2.0

2 The Implementation of GARUDA 2.0

2.1 Syntax

2.1.1 Definitions

2.1.2 Syntax of Predicates

2.1.3 Syntax of Policies

$$
\begin{aligned}
[\![\cdot]\!] \;:\; & \mathbf{Stream}(Eo) \times \mathbf{Stream}(Mi) \to \\
& P(\mathbf{Stream}(Eo)) \times P(\mathbf{Stream}(Mi)) \\
[\![0]\!](-,-) &\triangleq (\varnothing, \varnothing) \\
[\![1]\!](os, is) &\triangleq (\{os\}, \{is\}) \\
[\![f = n]\!](os, is) &\triangleq (\mathsf{filter}\ (f = n)\ \{os\},\ \mathsf{filter}\ (f = n)\ \{is\}) \\
[\![a + b]\!](os, is) &\triangleq [\![a]\!](os, is) \cup [\![b]\!](os, is) \\
& \quad \mathsf{where}\ (S_o^1, S_i^1) \cup (S_o^2, S_i^2) \triangleq (S_o^1 \cup S_o^2, S_i^1 \cup S_i^2) \\
[\![a \cdot b]\!](os, is) &\triangleq [\![a]\!](os, is) \cap [\![b]\!](os, is) \\
& \quad \mathsf{where}\ (S_o^1, S_i^1) \cap (S_o^2, S_i^2) \triangleq (S_o^1 \cap S_o^2, S_i^1 \cap S_i^2) \\
[\![\neg a]\!](os, os) &\triangleq \mathsf{let}\ (S_o, S_i) = [\![a]\!](os, is) \\
& \quad \mathsf{in}\ (\{os\} - S_o, \{is\} - S_i)
\end{aligned}
$$

Figure 4: The semantics of predicates in GARUDA 2.0. While predicates remain unchanged from GARUDA, the theory changes slightly

2.2 Semantics

2.2.1 Semantics of Predicates

2.2.2 Semantics of Policies

$$
\begin{aligned}
[\![\cdot]\!] \;:\; & \mathbf{Stream}(Eo) \times \mathbf{Stream}(Mi) \to \\
& P(\mathbf{Stream}(Eo)) \times P(\mathbf{Stream}(Mi)) \\
[\![(\Phi_{Encrypt}, \Phi_{Decrypt})]\!] \triangleq\; & \mathsf{let}\ (os \to os' = \Phi_{Encrypt}),\ (is \to is' = \Phi_{Decrypt}) \\
& \mathsf{in}\ (\{os'\}, \{is'\}) \\
[\![inj_{Eo}(o)]\!](os, is) \triangleq\; & (\{o : os\}, \{is\}) \\
[\![inj_{Mi}(i)]\!](os, is) \triangleq\; & (\{os\}, \{i : is\}) \\
[\![f \leftarrow n]\!](os, is) \triangleq\; & (\mathsf{map}\ (f \leftarrow n)\ \{os\},\ \mathsf{map}\ (f \leftarrow n)\ \{is\}) \\
[\![p + q]\!](os, is) \triangleq\; & [\![p]\!](os, is) \cup [\![q]\!](os, is) \\
& \mathsf{where}\ (S_o^1, S_i^1) \cup (S_o^2, S_i^2) \triangleq (S_o^1 \cup S_o^2, S_i^1 \cup S_i^2) \\
[\![p \cdot q]\!](os, is) \triangleq\; & \mathsf{let}\ (S_o, S_i) = [\![p]\!](os, is) \\
& \mathsf{in}\ \bigcup \{[\![q]\!](os', is') \mid os' \in S_o, is' \in S_i\}
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathsf{filter}\ f]\!](S) \triangleq\; & \{l \in S \mid f(l) = \mathsf{true}\} \\
[\![\mathsf{map}\ g]\!](S) \triangleq\; & \{g(l) \mid l \in S\}
\end{aligned}
$$

Figure 5: The semantics of policies in GARUDA 2.0.

Values	v	::=	INSTR | RES	
Registers	b	::=	reg	
Expressions	e	::=	$read(b)$	*Read Reg*
		|	$write(b, v)$	*Write Value to Reg*
		|	$let\ x = e_1\ in\ e_2$	*Assignment*
		|	$e_1 \mid\mid e_2$	*Parallel*
		|	$f(e_1)$	*Apply Function*
		|	$if\ v = n\ then\ e_1\ else\ e_2$	*Conditional*
		|	e_1 && e_2	*Product (AND)*
		|	$e_1; e_2$	*Concatenation*
		|	$forever\ e$	*Hardwire*

Figure 6: The language GARUDA 2.0 uses as a means to facilitate compilation.

PORT	DESCRIPTION
EXE	Direct output of the execution stage.
EXE'	Altered execution output.
	Input for state register.
MEM	Direct output of the state register.
MEM'	Altered memory access input.

Table 1: The four ports used in compilation. These are the inputs and outputs for logic between the functional units and state register.

3 Compilation to Verilog

3.1 The Intermediate Language

3.2 Compiling GARUDA 2.0 to the Intermediate

We define a function **C** to be the compilation of some predicate or policy in GARUDA 2.0 to the intermediate language. **C** operates on the four ports of the GARUDA 2.0 monitor. These consist of the processing done after the Execution stage, before the Memory stage, and their respective altered streams. Table 1 summarizes each port. Assume all **C** are short for $\mathbf{C}_{(EXE,EXE',MEM,MEM')}$ if unspecified.

3.2.1 Compiling Predicates

3.2.2 Compiling Policies

$$
\begin{aligned}
\mathbf{C}[0] &= let\ E\_bog = \text{new buf} \\
&\qquad M\_bog = \text{new buf} \\
&\quad in\ write(EXE', E\_bog) \\
&\qquad write(MEM', M\_bog) \\
\mathbf{C}[1] &= write(EXE', read(EXE)) \\
&\quad write(MEM', read(MEM)) \\
\mathbf{C}[f = n] &= if\ EXE = n\ then \\
&\qquad write(EXE', read(EXE)) \\
&\qquad else\ \mathbf{C}_{(EXE, EXE', -, -)}[0] \\
&\quad if\ MEM = n\ then \\
&\qquad write(MEM', read(MEM)) \\
&\qquad else\ \mathbf{C}_{(-, -, MEM, MEM')}[0] \\
\mathbf{C}[\mathsf{test}(a + b)] &= (*Intermediate\ Execution\ Ports*) \\
&\quad let\ E_{ai}, E'_{ai}, E_{bi}, E'_{bi} = \text{new buf}\ in \\
&\quad (*Intermediate\ Memory\ Ports*) \\
&\quad let\ M_{ai}, M'_{ai}, M_{bi}, M'_{bi} = \text{new buf}\ in \\
&\qquad DeMux(EXE, E_{ai}, E_{bi})\ || \\
&\qquad DeMux(MEM, M_{ai}, M_{bi}) \\
&\quad let\ i_a = \mathbf{C}_{(E_{ai}, E'_{ai}, M_{ai}, M'_{ai})}[a] \\
&\qquad i_b = \mathbf{C}_{(E_{bi}, E'_{bi}, M_{bi}, M'_{bi})}[b] \\
&\quad in\ Mux(i_a\ \ i_b, (EXE', MEM')) \\
\mathbf{C}[\mathsf{test}(a \cdot b)] &= \mathbf{C}[\mathsf{test}(a) \cdot \mathsf{test}(b)] \\
\mathbf{C}[\neg a] &= if\ \neg a\ then \\
&\qquad write(EXE', read(EXE)) \\
&\qquad write(MEM', read(MEM)) \\
&\quad else\ \mathbf{C}[0]
\end{aligned}
$$

Figure 7: Compilation of GARUDA 2.0 predicates into the intermediate language. These remain largely unchanged from GARUDA.

$$
\begin{aligned}
\mathbf{C}[inj_i V] &= write(i_{out}, V) \\
\mathbf{C}[inj_r V] &= write(r_{out}, V) \\
\mathbf{C}[f \leftarrow n] &= let\ i' = f(read(i_{in})) \\
&\qquad\quad r' = f(read(r_{in})) \\
&\quad in\ write(i_{out}, i') \\
&\qquad\quad write(r_{out}, r') \\
\mathbf{C}[p + q] &= let\ e_p = \mathbf{C}[p] \\
&\qquad\quad e_q = \mathbf{C}[q] \\
&\quad in\ Mux(e_p, e_q, (i_{out}, r_{out})) \\
\mathbf{C}[p \cdot q] &= let\ i_{mid} = \text{new buf} \\
&\qquad\quad r_{mid} = \text{new buf} \\
&\qquad\quad e_p = \mathbf{C}_{(i_{in}, i_{mid}, r_{in}, r_{mid})}[p] \\
&\qquad\quad e_q = \mathbf{C}_{(i_{mid}, i_{out}, r_{mid}, r_{out})}[q] \\
&\quad in\ e_p\ ;\ e_q
\end{aligned}
$$

Figure 8: Compilation of GARUDA 2.0 policies into the intermediate language.

4 Applications of GARUDA 2.0

4.1 Standard Taint

In the previous version of GARUDA, taint was implemented by tagging the most significant bit of a register. This caused no hardware overhead to maintain in the pipeline, but imposed an obvious bit-resolution hindrance. In GARUDA 2.0, the definition of such a policy is identical, but the compilation is different.

We assume a MIPS-like architecture for this example. The op codes of any given instruction are no more than 3; IN_1, IN_2, and OUT We denote these as RS, RT, and RD.

Let's suppose you want your taint to propogate on the inputs of any arithmetic instructions. Additionally, you prohibit any tainted instructions or addresses from accessing memory. In GARUDA 2.0, we can define this as follows. Please note that ALU can further be expanded to specific types of ALU instructions

$$
\begin{array}{rcl}
\textit{Instruction Fields } f_i & ::= & \mathsf{Taint_{RS}}, \mathsf{Taint_{RT}}, \mathsf{Taint_{RD}}, \mathsf{OP} \\
\mathsf{OP} & ::= & \mathsf{MEM_{READ}} \mid \mathsf{MEM_{WRITE}} \mid \mathsf{ALU} \mid \cdots \\
\textit{Result Fields } f_r & ::= & (*\mathsf{empty}*) \\
 & & \\
\mathsf{Arith} & \triangleq & \mathsf{OP} = \mathsf{ALU} \\
\mathsf{Read} & \triangleq & \mathsf{OP} = \mathsf{MEM_{READ}} \\
\mathsf{Write} & \triangleq & \mathsf{OP} = \mathsf{MEM_{WRITE}} \\
\mathsf{AnyMem} & \triangleq & \mathsf{Read} + \mathsf{Write} \\
 & & \\
\mathsf{TaintedInstr} & \triangleq & (\mathsf{Taint_{RS}} = \mathsf{TRUE}) + (\mathsf{Taint_{RT}} = \mathsf{TRUE}) \\
\mathsf{TaintRes} & \triangleq & \mathsf{Taint_{RD}} \leftarrow \mathsf{TRUE} \\
 & & \\
\mathsf{PropTaintALU} & \triangleq & act(\mathsf{Arith} \cdot \mathsf{TaintedInstr} \cdot \mathsf{TaintRes}) \\
 & & +act(\mathsf{Arith} \cdot \neg\mathsf{TaintedInstr}) \\
\mathsf{NoTaintMem} & \triangleq & act(\mathsf{AnyMem} \cdot \neg\mathsf{TaintedInstr}) \\
\mathsf{SecureMem} & \triangleq & \mathsf{PropTaintALU} + \mathsf{NoTaintMem} \\
 & & +act(\neg(\mathsf{Arith} + \mathsf{AnyMem}))
\end{array}
$$

4.2 Speculative Taint

[1]

$$
\begin{array}{rcl}
\textit{Instruction Fields } f_i & ::= & \mathsf{Taint_{RS}}, \mathsf{Taint_{RT}}, \mathsf{Taint_{RD}}, \mathsf{OP} \\
\mathsf{OP} & ::= & \mathsf{MEM_{READ}} \mid \mathsf{MEM_{WRITE}} \mid \mathsf{ALU} \mid \cdots \\
\textit{Result Fields } f_r & ::= & (*\mathsf{empty}*) \\
 & & \\
\mathsf{Arith} & \triangleq & \mathsf{OP} = \mathsf{ALU} \\
\mathsf{Read} & \triangleq & \mathsf{OP} = \mathsf{MEM_{READ}} \\
\mathsf{Write} & \triangleq & \mathsf{OP} = \mathsf{MEM_{WRITE}} \\
\mathsf{AnyMem} & \triangleq & \mathsf{Read} + \mathsf{Write} \\
 & & \\
\mathsf{TaintedInstr} & \triangleq & (\mathsf{Taint_{RS}} = \mathsf{TRUE}) + (\mathsf{Taint_{RT}} = \mathsf{TRUE}) \\
\mathsf{TaintRes} & \triangleq & \mathsf{Taint_{RD}} \leftarrow \mathsf{TRUE} \\
 & & \\
\mathsf{PropTaintALU} & \triangleq & act(\mathsf{Arith} \cdot \mathsf{TaintedInstr} \cdot \mathsf{TaintRes}) \\
 & & +act(\mathsf{Arith} \cdot \neg\mathsf{TaintedInstr}) \\
\mathsf{NoTaintMem} & \triangleq & act(\mathsf{AnyMem} \cdot \neg\mathsf{TaintedInstr}) \\
\mathsf{SecureMem} & \triangleq & \mathsf{PropTaintALU} + \mathsf{NoTaintMem} \\
 & & +act(\neg(\mathsf{Arith} + \mathsf{AnyMem}))
\end{array}
$$

References

[1] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. Speculative taint tracking (stt) a comprehensive protection for speculatively accessed data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 954–968, 2019.