GARUDA 2.0

Sage Sefton
ss557415@ohio.edu
Ohio University

Avinash Karanth
karanth@ohio.edu
Ohio University

2020
Fall

$$
\begin{array}{llll}
\text{EXE Input Fields} & f_{Ei} & ::= & f_{Ei_1} \mid \cdots \mid f_{Ei_k} \\
\text{State Reg Fields} & f_{SR} & ::= & f_{SR_1} \mid \cdots \mid f_{SR_k} \\
\text{MEM Input Fields} & f_{Mi} & ::= & f_{Mi_1} \mid \cdots \mid f_{Mi_k} \\
\text{Fields} & f & ::= & f_{Ei} \mid f_{SR} \mid f_{Mi} \\
\text{Obfuscated Field} & f^{\mathbf{o}} & ::= & f \\
\text{Inputs} & i & ::= & \{ f_{i_1} = v_{i_1}, \ \ldots \ , \ f_{i_k} = v_{i_k} \} \\
\text{Outputs} & o & ::= & \{ f_{o_1} = v_{o_1}, \ \ldots \ , \ f_{o_k} = v_{o_k} \} \\
\text{Obfuscation Fxn} & \Phi & ::= & f \rightarrow f^{\mathbf{o}} \mid f^{\mathbf{o}} \rightarrow f \\
\end{array}
$$

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.

2

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

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

$$\begin{array}{rlll}
\text{Policies} \quad p, q \quad ::= & \textbf{test}(a) & & \textit{Test} \\
& \mid & (\Phi_{Encrypt}, \Phi_{Decrypt}) & \textit{Obfuscation} \\
& \mid & inj_{SR} & \textit{Injection to State Register} \\
& \mid & inj_{Mi} & \textit{Injection to Memory Unit} \\
& \mid & f \leftarrow n & \textit{Update} \\
& \mid & p + q & \textit{Choice} \\
& \mid & p \cdot q & \textit{Sequential Concatenation}
\end{array}$$

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

$$\llbracket \cdot \rrbracket \ : \ \mathbf{Stream}(E) \times \mathbf{Stream}(M) \to$$
$$P(\mathbf{Stream}(E)) \times P(\mathbf{Stream}(M))$$
$$\llbracket 0 \rrbracket (-, -) \triangleq (\varnothing, \varnothing)$$
$$\llbracket 1 \rrbracket (es, ms) \triangleq (\{es\}, \{ms\})$$
$$\llbracket f = n \rrbracket (es, ms) \triangleq (\mathsf{filter} \ (f = n) \ \{es\}, \ \mathsf{filter} \ (f = n) \ \{ms\})$$
$$\llbracket a + b \rrbracket (es, ms) \triangleq \llbracket a \rrbracket (es, ms) \cup \llbracket b \rrbracket (es, ms)$$
$$\mathsf{where} \ (S_e^1, S_m^1) \cup (S_e^2, S_m^2) \triangleq (S_e^1 \cup S_e^2, S_m^1 \cup S_m^2)$$
$$\llbracket a \cdot b \rrbracket (es, ms) \triangleq \llbracket a \rrbracket (es, ms) \cap \llbracket b \rrbracket (es, ms)$$
$$\mathsf{where} \ (S_e^1, S_m^1) \cap (S_e^2, S_m^2) \triangleq (S_e^1 \cap S_e^2, S_m^1 \cap S_m^2)$$
$$\llbracket \neg a \rrbracket (es, ms) \triangleq \mathsf{let} \ (S_e, S_m) = \llbracket a \rrbracket (es, ms)$$
$$\mathsf{in} \ (\{es\} - S_e, \{ms\} - S_m)$$

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

$$\llbracket \cdot \rrbracket \ : \ \mathbf{Stream}(E) \times \mathbf{Stream}(M) \rightarrow$$
$$P(\mathbf{Stream}(E)) \times P(\mathbf{Stream}(M))$$

$$\llbracket (\Phi_{Encrypt}, \Phi_{Decrypt}) \rrbracket (es, ms) \ \triangleq \ \mathsf{let} \ (e \rightarrow e^{\mathbf{o}} = \Phi_{Encrypt}), \ (m^{\mathbf{o}} \rightarrow m = \Phi_{Decrypt})$$
$$\mathsf{in} \ (\{e^{\mathbf{o}}\}, \{m\})$$

$$\llbracket inj_{SR}(e) \rrbracket (es, ms) \triangleq (\{e : es\}, \{ms\})$$

$$\llbracket inj_{Mi}(m) \rrbracket (es, ms) \triangleq (\{es\}, \{m : ms\})$$

$$\llbracket f \leftarrow n \rrbracket (es, ms) \triangleq (\mathsf{map} \ (f \leftarrow n) \ \{e\}, \ \mathsf{map} \ (f \leftarrow n) \ \{m\})$$

$$\llbracket p + q \rrbracket (es, ms) \triangleq \llbracket p \rrbracket (e, m) \cup \llbracket q \rrbracket (e, m)$$
$$\mathsf{where} \ (S_e^1, S_m^1) \cup (S_e^2, S_m^2) \triangleq (S_e^1 \cup S_e^2, S_m^1 \cup S_m^2)$$

$$\llbracket p \cdot q \rrbracket (es, ms) \triangleq \mathsf{let} \ (S_e, S_m) = \llbracket p \rrbracket (e, m)$$
$$\mathsf{in} \ \bigcup \{ \llbracket q \rrbracket (e', m') \mid e' \in S_e, m' \in S_m \}$$

$$\llbracket \mathsf{filter} \ f \rrbracket (S) \triangleq \{ l \in S \mid f(l) = \mathsf{true} \}$$
$$\llbracket \mathsf{map} \ g \rrbracket (S) \triangleq \{ g(l) \mid l \in S \}$$

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
			$e_1\ \|\|\ e_2$	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.
$EXE^\mathbf{o}$	Altered state register input.
$MEM^\mathbf{o}$	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 $\mathbf{C}_{(EXE,EXE^\mathbf{o},MEM^\mathbf{o},MEM)}$ if unspecified.

3.2.1   Compiling Predicates

3.2.2   Compiling Policies

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

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

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

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 com-
pilation 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

$$
\begin{aligned}
\textit{Instruction Fields } f_i \quad &::= \quad \mathsf{Taint_{RS}}, \mathsf{Taint_{RT}}, \mathsf{Taint_{RD}}, \mathsf{OP} \\
\mathsf{OP} \quad &::= \quad \mathsf{MEM_{READ}} \mid \mathsf{MEM_{WRITE}} \mid \mathsf{ALU} \mid \cdots \\
\textit{Result Fields } f_r \quad &::= \quad (*\mathsf{empty}*)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{Arith} \quad &\triangleq \quad \mathsf{OP} = \mathsf{ALU} \\
\mathsf{Read} \quad &\triangleq \quad \mathsf{OP} = \mathsf{MEM_{READ}} \\
\mathsf{Write} \quad &\triangleq \quad \mathsf{OP} = \mathsf{MEM_{WRITE}} \\
\mathsf{AnyMem} \quad &\triangleq \quad \mathsf{Read} + \mathsf{Write}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{TaintedInstr} \quad &\triangleq \quad (\mathsf{Taint_{RS}} = \mathsf{TRUE}) + (\mathsf{Taint_{RT}} = \mathsf{TRUE}) \\
\mathsf{TaintRes} \quad &\triangleq \quad \mathsf{Taint_{RD}} \leftarrow \mathsf{TRUE}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{PropTaintALU} \quad &\triangleq \quad act(\mathsf{Arith} \cdot \mathsf{TaintedInstr} \cdot \mathsf{TaintRes}) \\
&\qquad + act(\mathsf{Arith} \cdot \neg\mathsf{TaintedInstr}) \\
\mathsf{NoTaintMem} \quad &\triangleq \quad act(\mathsf{AnyMem} \cdot \neg\mathsf{TaintedInstr}) \\
\mathsf{SecureMem} \quad &\triangleq \quad \mathsf{PropTaintALU} + \mathsf{NoTaintMem} \\
&\qquad + act(\neg(\mathsf{Arith} + \mathsf{AnyMem}))
\end{aligned}
$$

4.2   Speculative Taint

[1]

$$
\begin{aligned}
\textit{Instruction Fields } f_i \quad &::= \quad \mathsf{Taint_{RS}}, \mathsf{Taint_{RT}}, \mathsf{Taint_{RD}}, \mathsf{OP} \\
\mathsf{OP} \quad &::= \quad \mathsf{MEM_{READ}} \mid \mathsf{MEM_{WRITE}} \mid \mathsf{ALU} \mid \cdots \\
\textit{Result Fields } f_r \quad &::= \quad (*\mathsf{empty}*)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{Arith} \quad &\triangleq \quad \mathsf{OP} = \mathsf{ALU} \\
\mathsf{Read} \quad &\triangleq \quad \mathsf{OP} = \mathsf{MEM_{READ}} \\
\mathsf{Write} \quad &\triangleq \quad \mathsf{OP} = \mathsf{MEM_{WRITE}} \\
\mathsf{AnyMem} \quad &\triangleq \quad \mathsf{Read} + \mathsf{Write}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{TaintedInstr} \quad &\triangleq \quad (\mathsf{Taint_{RS}} = \mathsf{TRUE}) + (\mathsf{Taint_{RT}} = \mathsf{TRUE}) \\
\mathsf{TaintRes} \quad &\triangleq \quad \mathsf{Taint_{RD}} \leftarrow \mathsf{TRUE}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{PropTaintALU} \quad &\triangleq \quad act(\mathsf{Arith} \cdot \mathsf{TaintedInstr} \cdot \mathsf{TaintRes}) \\
&\qquad + act(\mathsf{Arith} \cdot \neg\mathsf{TaintedInstr}) \\
\mathsf{NoTaintMem} \quad &\triangleq \quad act(\mathsf{AnyMem} \cdot \neg\mathsf{TaintedInstr}) \\
\mathsf{SecureMem} \quad &\triangleq \quad \mathsf{PropTaintALU} + \mathsf{NoTaintMem} \\
&\qquad + act(\neg(\mathsf{Arith} + \mathsf{AnyMem}))
\end{aligned}
$$

11

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.