GARUDA 2.0

Sage Sefton
ss557415@ohio.edu
Ohio University

Avinash Karanth
karanth@ohio.edu
Ohio University

2020
Spring

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	::=	$f_{Ei} \mid f_{Eo} \mid f_{Mi}$
Instructions	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	λ	::=	$f \to f'$

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.

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

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

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

Figure 4: The semantics of predicates in GARUDA 2.0.

2.2 Semantics

2.2.1 Semantics of Predicates

2.2.2 Semantics of Policies

$$
\begin{aligned}
[\![inj_i(i)]\!](is, rs) &\triangleq (\{i : is\}, \{rs\}) \\
[\![inj_r(r)]\!](is, rs) &\triangleq (\{is\}, \{r : rs\}) \\
[\![f \leftarrow n]\!](is, rs) &\triangleq (\mathsf{map}\ (f \leftarrow n)\ \{is\},\ \mathsf{map}\ (f \leftarrow n)\ \{rs\}) \\
[\![p + q]\!](is, rs) &\triangleq [\![p]\!](is, rs) \cup [\![q]\!](is, rs) \\
&\quad \mathsf{where}\ (S_i^1, S_r^1) \cup (S_i^2, S_r^2) \triangleq (S_i^1 \cup S_i^2, S_r^1 \cup S_r^2) \\
[\![p \cdot q]\!](is, rs) &\triangleq \mathsf{let}\ (S_i, S_r) = [\![p]\!](is, rs) \\
&\quad \mathsf{in}\ \bigcup \{[\![q]\!](is', rs') \mid is' \in S_i, rs' \in S_r\}
\end{aligned}
$$

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

Figure 5: The semantics of policies in GARUDA 2.0.

3 Intermediate Syntax

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)*
		|	$e_1; e_2$	*Concatination*
		|	$forever\ e$	*Hardwire*

4 Intermediate Semantics

Assume all $\mathbf{C}$ is short for $\mathbf{C}_{(i_{in},i_{out},r_{in},r_{out})}$ if unspecified.

$$
\begin{aligned}
\mathbf{C}[0] &= \mathit{let}\ i\_bog = \text{new buf} \\
&\qquad\quad r\_bog = \text{new buf} \\
&\quad \mathit{in}\ \mathit{write}(i_{out}, i\_bog) \\
&\qquad \mathit{write}(r_{out}, r\_bog) \\
\mathbf{C}[1] &= \mathit{write}(i_{out}, \mathit{read}(i_{in})) \\
&\quad \mathit{write}(r_{out}, \mathit{read}(r_{in})) \\
\mathbf{C}[f = n] &= \mathit{if}\ i_{in} = n\ \mathit{then} \\
&\qquad \mathit{write}(i_{out}, \mathit{read}(i_{in})) \\
&\qquad \mathit{else}\ \mathbf{C}_{(i_{in},i_{out},-,-)}[0] \\
&\quad \mathit{if}\ r_{in} = n\ \mathit{then} \\
&\qquad \mathit{write}(r_{out}, \mathit{read}(r_{in})) \\
&\qquad \mathit{else}\ \mathbf{C}_{(-,-,r_{in},r_{out})}[0]
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{C}[\mathsf{test}(a+b)] &= let\ e_{aii}, e_{ari}, e_{bii}, e_{bri} = \text{new buf}\ in \\
&\quad DeMux(i_{in}, e_{aii}, e_{bii}) \mid\mid DeMux(r_{in}, e_{ari}, e_{bri}) \\
&\quad let\ e_a = \mathbf{C}_{(e_{aii}, e_{aio}, e_{ari}, e_{aro})}[a] \\
&\qquad e_b = \mathbf{C}_{(e_{bii}, e_{bio}, e_{bri}, e_{bro})}[b] \\
&\quad in\ Mux(e_a \ \ e_b, (i_{out}, r_{out})) \\
\mathbf{C}[\mathsf{test}(a \cdot b)] &= \mathbf{C}[\mathsf{test}(a) \cdot \mathsf{test}(b)] \\
\mathbf{C}[\neg a] &= if\ \neg a\ then \\
&\qquad write(i_{out}, read(i_{in})) \\
&\qquad write(r_{out}, read(r_{in})) \\
&\quad else\ \mathbf{C}[0] \\
\mathbf{C}[act(p)] &= let\ r\_bog_i = \text{new buf} \\
&\qquad r\_bog_o = \text{new buf} \\
&\quad in\ e_p = \mathbf{C}_{(i_{in}, i_{out}, r\_bog_i, r\_bog_o)}[p] \\
&\quad e_p \mid\mid write(r_{out}, read(r_{in})) \\
\mathbf{C}[res(p)] &= let\ i\_bog_i = \text{new buf} \\
&\qquad i\_bog_o = \text{new buf} \\
&\quad in\ e_p = \mathbf{C}_{(i\_bog_i, i\_bog_o, r_{in}, r_{out})}[p] \\
&\quad e_p \mid\mid write(i_{out}, read(i_{in})) \\
\mathbf{C}[inj_i V] &= write(i_{out}, V) \\
\mathbf{C}[inj_r V] &= write(r_{out}, V) \\
\mathbf{C}[f \leftarrow n] &= let\ i' = f(read(i_{in})) \\
&\qquad r' = f(read(r_{in})) \\
&\quad in\ write(i_{out}, i') \\
&\qquad write(r_{out}, r') \\
\mathbf{C}[p+q] &= let\ e_p = \mathbf{C}[p] \\
&\qquad e_q = \mathbf{C}[q] \\
&\quad in\ Mux(e_p, e_q, (i_{out}, r_{out})) \\
\mathbf{C}[p \cdot q] &= let\ i_{mid} = \text{new buf} \\
&\qquad r_{mid} = \text{new buf} \\
&\qquad e_p = \mathbf{C}_{(i_{in}, i_{mid}, r_{in}, r_{mid})}[p] \\
&\qquad e_q = \mathbf{C}_{(i_{mid}, i_{out}, r_{mid}, r_{out})}[q] \\
&\quad in\ e_p\ ;\ e_q
\end{aligned}
$$

5 Applications of GARUDA 2.0

5.1 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

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

5.2 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.