15-316: Software Foundations of Security and Privacy

Lecture Notes on Control Flow Safety & Security Automata

Matt Fredrikson

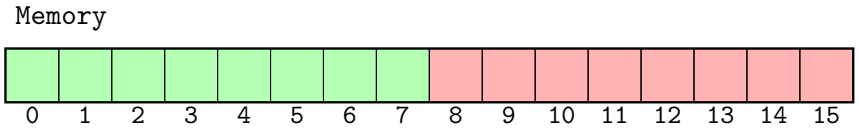
Carnegie Mellon University
Lecture 8

1 Introduction & Recap

In the last lecture we talked about enforcing a more granular type of memory safety policy to ensure that parts of our program don't read or write portions they aren't supposed to. This was motivated by our hypothetical career as an app developer who wants to monetize with advertising, and is thus compelled by Vladimir's discount ad shop to run untrusted rendering code within our program:

$$\texttt{if}(\textit{display ads})\ \alpha\ \texttt{else}\ \textit{continue without ads}$$

We discussed sandboxing policies where a region of memory is designated for the untrusted α to "play" in, such as the upper portion of memory at addresses 8-15 in the diagram below.



As long as we can enforce this policy, and we are careful about writing our program to save and restore variable state, then we can ensure that whatever the sandbox does will not affect the rest of our program's execution.

We discussed an approach called *software fault isolation* [3, 5] (SFI) for properly isolating the malicious or buggy effects of α from the rest of our program. SFI works by inlining enforcement directly into α, changing its behavior so that it can't violate the sandbox policy and if it attempts to do so then it still won't have any effect on the rest of our execution. SFI is a very practical technique, and has been used effectively in real

applications to isolate untrusted code execution from browsers, operating systems, and other critical applications. In the next lab, you will implement a prototype SFI policy for your server.

Today we will look at a related technique called *control flow integrity* [1], which ensures that the attacker cannot influence the control flow of a program to diverge from a pre-defined control flow policy. But in order for this defense to have any purpose, we need to introduce indirect control flow commands into our language, bringing it closer yet to the features that real platforms in need of rigorous security defenses have in practice. We will then generalize the safety enforcement techniques discussed so far, introducing a flexible and practical method for enforcing safety policies provided as *security automata* [2].

2 Indirect control flow

So far the programs that we have considered have a particularly nice property. Namely, once the programmer decides which commands are in the program and how they are sequenced together with compositions, conditionals, and while loops, then all of the possible sequences of commands that the program will ever execute are fixed once and for all. There is no way for a user to provide data that could cause some of the commands to be skipped over or added, and as long as the program is executed faithfully to the semantics, the control flow will be as the programmer envisioned when the program was written.

Programs executed on "real" machines do not enjoy this property, thanks to indirect transfers of control flow. An indirect control flow transfer is commonly implemented using a function pointer in high-level languages, or a `jmp` instruction with a pointer operand. We'll add this functionality to our language by considering a program command of the form:

$$\texttt{if}(Q)\,\texttt{jump}\,e \tag{1}$$

The command in (1) first tests whether a formula Q is true in the current state. If it is, then control transfers to the instruction indexed by the term e. Otherwise, control proceeds to the next instruction.

But this doesn't make much since yet, because we haven't discussed indexing of instructions. Programs in the simple imperative language are themselves just commands, which can be built from other commands by connecting them with composition, conditional, and looping constructs. We will now change the language so that rather than having structured high-level commands like $\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta$ and $\texttt{while}(Q)\,\alpha$, we will assume that programs are sequences of unstructured "atomic" commands. So the commands are defined by the syntax:

$$\alpha \;::=\; x := e \mid \texttt{Mem}(e) := \tilde{e} \mid \texttt{assert}(Q) \mid \texttt{if}(Q)\,\texttt{jump}\,e$$

Then a program Π is a finite sequence of commands,

$$\Pi = (\alpha_0, \alpha_1, \ldots, \alpha_n) \tag{2}$$

We will write $\Pi(i)$, where $0 \leq i \leq n$, to refer to the command α_i in program Π. If i is negative, or $n < i$, then $\Pi(i)$ is undefined.

Think of this language as a simplified version of assembly language. Memory update commands $\texttt{Mem}(e) := \tilde{e}$ correspond to store instructions (i.e., `mov` into a memory cell), memory dereference terms $\texttt{Mem}(e)$ correspond to memory fetch instructions (i.e., `mov` from a memory cell to a register), and $\texttt{if}(Q)\ \texttt{jump}\ e$ to conditional `jmp` instructions. We don't have an explicit stack or notion of procedure to worry about, but if we did then `halt` commands would correspond to `ret` instructions.

Semantics. Recall that program states ω are composed of a variable map ω_V and memory ω_M. Now that programs Π are composed of indexed commands, and can transfer control to any command in Π, states will also need to track a program counter ω_i that determines which command executes next. The program counter will range from $i \in 0$ to n, denoting that the command Π_i executes next.

Definition 1 (Operational semantics of programs)**.** The small-step transition relation $\rightsquigarrow_\Pi$ of program Π composed of commands $\alpha_0, \ldots, \alpha_n$ in state $\omega = (\omega_i, \omega_V, \omega_M)$ is given by the following cases:

$$(\omega_i, \omega_V, \omega_M) \rightsquigarrow_\Pi \begin{cases} (\omega_i + 1, \omega_V\{x \mapsto \omega[\![e]\!]\}, \omega_M) & \textit{if}\ \ \Pi_i = x := e \text{ and } \omega[\![e]\!] \text{ is defined} \\ (\omega_i + 1, \omega_V, \omega_M\{\omega[\![e]\!] \mapsto \omega[\![\tilde{e}]\!]\}) & \textit{if}\ \ \Pi_i = \texttt{Mem}(e) := \tilde{e} \text{ and} \\ & \quad \omega[\![\tilde{e}]\!] \text{ is defined and } 0 \leq e < U \\ (\omega_i + 1, \omega_V, \omega_M) & \textit{if}\ \ \Pi_i = \texttt{assert}(Q) \text{ and } \omega \models Q \\ (\omega[\![e]\!], \omega_V, \omega_M) & \textit{if}\ \ \Pi_i = \texttt{if}(Q)\ \texttt{jump}\ e \text{ and} \\ & \quad 0 \leq \omega[\![e]\!] \leq n \text{ and } \omega \models Q \\ (\omega_i + 1, \omega_V, \omega_M) & \textit{if}\ \ \Pi_i = \texttt{if}(Q)\ \texttt{jump}\ e \text{ and} \\ & \quad 0 \leq \omega[\![e]\!] \leq n \text{ and } \omega \models \neg Q \\ \Lambda & \textit{if}\ \ \text{otherwise and } 0 \leq \omega_i \leq n \end{cases}$$

Having defined the small-step transition semantics, we can define the trace semantics as all sequences of states $\omega_1, \omega_2, \ldots$ that either terminate, diverge (i.e. terminate in no state and run forever), or abort by terminating in $\omega = \Lambda$.

Definition 2 (Trace semantics)**.** Given a program Π composed of commands $\alpha_0, \ldots, \alpha_n$, the trace semantics $[\![\Pi]\!]$ is the set of traces obtainable by repeated application of the small-step relation $\rightsquigarrow_\Pi$.

$$[\![\Pi]\!] = \{(\omega_0, \omega_1, \ldots) \ :\ \omega_i \rightsquigarrow_\Pi \omega_{i+1} \text{ for all indices } 0 \leq i \text{ of the trace}\}$$

The definitions of terminating, diverging, and aborting traces are the same as they were in previous definitions of the trace semantics.

Example. Consider the following program to illustrate how the operational and trace semantics work.

$$\begin{aligned}
&\texttt{0:}\quad \texttt{assert}(0 \leq x)\\
&\texttt{1:}\quad \texttt{Mem}(0) := \texttt{Mem}(0) + 1\\
&\texttt{2:}\quad x := x - 1\\
&\texttt{3:}\quad \texttt{if}(0 \leq x)\ \texttt{jump}\ 1
\end{aligned}$$

Suppose that we begin in the following state:

$$\begin{aligned}
&\omega_i = 0\\
&\omega_V(x) = 2 \text{ and all other variables map to } 0\\
&\omega_M(i) = 0 \text{ for all } 0 \leq i < U
\end{aligned}$$

Then consulting the operational semantics, we see that $\Pi_0 = \texttt{assert}(0 \leq x)$ and $\omega \models 0 \leq x$, so the next state is $(\omega_i + 1, \omega_V, \omega_M)$.

$$\begin{aligned}
&(0, \omega_V, \omega_M) \rightsquigarrow_\Pi\\
&(1, \omega_V, \omega_M)
\end{aligned}$$

Now $\Pi_1 = \texttt{Mem}(0) := \texttt{Mem}(0) + 1$ and $\texttt{Mem}(0) = 0$. So the next state is $(2, \omega_V, \omega_M\{0 \mapsto 1\})$.

$$\begin{aligned}
&(0, \omega_V, \omega_M) \rightsquigarrow_\Pi\\
&(1, \omega_V, \omega_M) \rightsquigarrow_\Pi\\
&(2, \omega_V, \omega_M\{0 \mapsto 1\})
\end{aligned}$$

Now $\Pi_2 = x := x - 1$ and the semantics tell us to update ω_V at x.

$$\begin{aligned}
&(0, \omega_V, \omega_M) \rightsquigarrow_\Pi\\
&(1, \omega_V, \omega_M) \rightsquigarrow_\Pi\\
&(2, \omega_V, \omega_M\{0 \mapsto 1\}) \rightsquigarrow_\Pi\\
&(3, \omega_V\{x \mapsto 0\}, \omega_M\{0 \mapsto 1\})
\end{aligned}$$

We now get to the jump because $\Pi_3 = \texttt{if}(0 \leq x)\ \texttt{jump}\ 1$. The number of instructions $n = 3$, so the next state has program counter 1. We continue in this way, until we reach the conditional jump again. At this point $\omega_V(x) = -1$, and so the program counter increments to 4.

$$\begin{aligned}
&(0, \omega_V, \omega_M) \rightsquigarrow_\Pi\\
&(1, \omega_V, \omega_M) \rightsquigarrow_\Pi\\
&(2, \omega_V, \omega_M\{0 \mapsto 1\}) \rightsquigarrow_\Pi\\
&(3, \omega_V\{x \mapsto 0\}, \omega_M\{0 \mapsto 1\}) \rightsquigarrow_\Pi\\
&(1, \omega_V\{x \mapsto 0\}, \omega_M\{0 \mapsto 1\}) \rightsquigarrow_\Pi\\
&(2, \omega_V\{x \mapsto 0\}, \omega_M\{0 \mapsto 1\}\{0 \mapsto 2\}) \rightsquigarrow_\Pi\\
&(3, \omega_V\{x \mapsto 0\}\{x \mapsto -1\}, \omega_M\{0 \mapsto 1\}\{0 \mapsto 2\}) \rightsquigarrow_\Pi\\
&(4, \omega_V\{x \mapsto 0\}\{x \mapsto -1\}, \omega_M\{0 \mapsto 1\}\{0 \mapsto 2\})
\end{aligned}$$

Now the program counter is outside the instruction bounds in Π. The operational semantics does not define a next state, so the computation effectively terminates.

3 Control Flow Integrity

Let's return to our problem with untrusted advertising code. Now that the language α is written in can make indirect jumps, what could go wrong? Assuming that we are using SFI to enforce a sandboxing policy, there is still no way for α to read or write memory outside the sandbox. Is this true? Consider the following situation, where we can assume that SFI has been applied to the untrusted α starting at command 20.

$$
\begin{array}{rl}
 & \vdots \\
\texttt{10:} & z := \texttt{Mem}(x) \\
\texttt{11:} & \texttt{if}(i \geq 0)\ \texttt{jump}\ y \\
 & \vdots \\
\alpha\ \left\{\begin{array}{rl}
\texttt{20:} & i := 0 \\
\texttt{21:} & x := \textit{attacker's desired address} \\
\texttt{22:} & y := 24 \\
\texttt{23:} & \texttt{if}(0 = 0)\ \texttt{jump}\ 10 \\
\texttt{24:} & \textit{copy memory contents from } z \\
 & \vdots
\end{array}\right.
\end{array}
$$

Here, our original program (not the untrusted α) dereferences memory and makes use of indirect control flow transfer. More specifically,

1. At command 10, it dereferences memory on the variable x and stores the result into z. Because this command is not in the untrusted portion α, it was not rewritten with SFI and can readily access memory outside the sandbox.

2. At command 11, the program tests $i \geq 0$, and if the test holds then jumps to whatever command is currently in y.

3. The untrusted code sets up the program state: *(i)* the indirect jump at 11 will occur by setting $i := 0$ on line 20; *(ii)* the indirect jump at line 11 will return control to α by setting $y := 24$; *(iii)* setting $x := \ldots$ at 21 so that the address read at line 10 will be whatever the attacker wants, presumably outside the sandbox bounds.

4. The command at 23 then executes an indirect jump on a trivial test, targeting 10 so that the attacker's choice of memory is read and control returns to α after the indirect jump at 11.

To summarize, the attacker identifies a sequence of commands in the trusted portion of the program, and sets things up in a way so that unauthorized memory is copied into a variable that the attacker can later access once control is returned to the untrusted code.

This should remind you of a *return-oriented programming* (ROP) attacks [4] that you learned about in 15-213. If we assume that the attacker knows the text of our program, then it is possible for them to identify "gadgets" in *code that we wrote* to do their bidding. But this crucially relies on the ability to change control flow using indirection so that commands are executed in the order needed by the attacker to carry out their goals.

3.1 Coarse-grained safety

How can we prevent this? One idea is to use the same approach as we did for SFI. In that case, we designed a sandbox between memory address s_l and s_h, and then rewrote the indices in all memory operations to ensure that accesses stayed within those bounds. Perhaps we can do a very similar thing here, by assigning a "code sandbox" between commands at $\texttt{pc}_l$ and $\texttt{pc}_h$. Then we can rewrite indirect jump commands to ensure that their target always lies within these bounds.

$$\textit{Rewrite all } \texttt{if}(Q)\,\texttt{jump}\,e \textit{ commands as } \texttt{if}(Q)\,\texttt{jump}\,(e\,\texttt{\&}\,\texttt{pc}_h)\mid\texttt{pc}_l \tag{3}$$

This is a form of *control flow integrity* (CFI) [1], a technique for enforcing a broad class of safety properties that place limits on the allowed control flow paths in a program. As long as we choose $\texttt{pc}_l$ and $\texttt{pc}_h$ to satisfy similar conditions as those used in Theorem **??**, i.e.

$$0 \le \texttt{pc}_l \le (e\,\texttt{\&}\,\texttt{pc}_h)\mid\texttt{pc}_l \le \texttt{pc}_h \le n \tag{4}$$

then we can ensure that the untrusted code will never jump out of its sandbox. This seems to address our concerns with the advertising scenario, when everything untrusted resides in a well-specified region of code known in advance.

3.2 Finer-grained control-flow safety

But what if this isn't the case? Suppose that we want to enforce other invariants on untrusted code, such as that they do not modify a protected variable under certain conditions. So for example if x is negative, then we want to jump over any assignment to x. We make the following replacements, among others:

$$\texttt{i:}\quad x := e \qquad \text{becomes} \qquad \begin{array}{rl}\texttt{i:} & \texttt{if}(x<0)\,\texttt{jump}\,i+2\\ \texttt{i+1:} & x := e\end{array}$$

Now if we use the coarse-grained CFI policy from before, can we actually enforce the policy using this approach? It would seem not, at least as long as the attacker knows that this is how we will attempt to do so. The problem arises because of the fact that according to the coarse-grained CFI policy, any address in the untrusted code is an allowed target of a jump. So it is perfectly acceptable (according to the coarse-grained policy) for the attacker to jump directly past the inlined check.

To see this more concretely, consider the following proof-of-concept attack code. Suppose that the attacker wants to set x to 0 regardless of what its value is before the untrusted code executes. This obviously violates the policy, and to accomplish it the

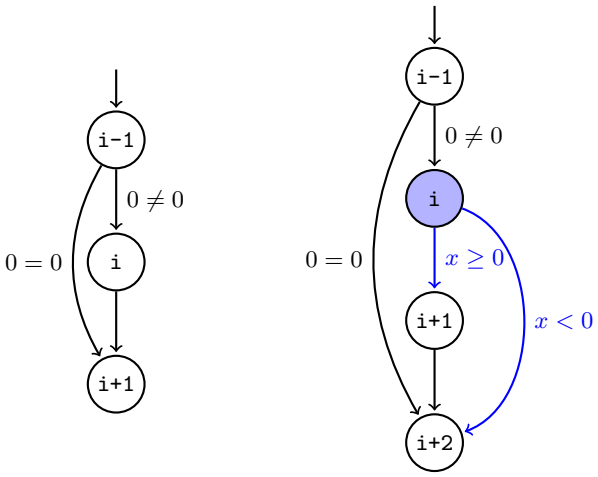
attacker will provide a program that takes the inline enforcement code into account. So after providing the code on the left,

$$\begin{array}{ll} \texttt{i-1:} & \texttt{if}(0=0)\,\texttt{jump}\,i+1 \\ \texttt{i:} & x := 0 \end{array} \qquad \text{becomes} \qquad \begin{array}{ll} \texttt{i-1:} & \texttt{if}(0=0)\,\texttt{jump}\,(i+1\,\&\,s_h)\mid s_l \\ \texttt{i:} & \texttt{if}(x<0)\,\texttt{jump}\,i+2 \\ \texttt{i+1:} & x := 0 \end{array}$$

In short, the attacker sets up the program to jump directly over the enforcement code.

Enforcing the control flow graph. To address this, we can rely on the control flow graph (CFG) of the untrusted code. Recall from lecture 5 that a program's CFG is a graph that encodes all of the possible valid transitions between commands in the program. In this case, we will obtain a control flow graph for the original untrusted program[1], and ensure that the program instrumented with inline safety checks follows the same CFG modulo any checks.

So in this case, the original control flow graph is given on the left of the diagram below. Obviously it is not the case that $0 \neq 0$, so the edge from `i-1` to `i` is never taken. After the invariant instrumentation is inserted, the correct translation of the control flow, preserving the relative edges from the original CFG, is shown on the right. The instrumentation replaced the instruction originally at `i` with an inline check, and shifted all of the subsequent instructions up by one address. So this moves `i` to `i+1`, and `i+1` to `i+2`. To make this clear in the diagram, the nodes and edges corresponding to instrumentation are marked in blue.



Now to correctly enforce the safety policy that the CFG on the right is respected by the

[1] Obtaining the CFG for an arbitrary program with indirect jumps is a difficult problem indeed. It may not always be possible to do so, and we will come back to this in later lectures. For now, we will just assume that we have obtained the correct CFG for α by some unknown means.

code, the original jumps are rewritten accordingly.

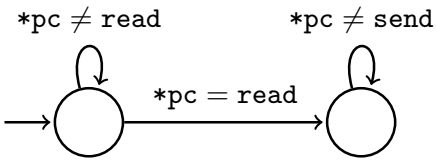
$$\begin{array}{rl} \texttt{i-1:} & \texttt{if}(0 = 0)\,\texttt{jump}\,i + 2 \\ \texttt{i:} & \texttt{if}(x < 0)\,\texttt{jump}\,i + 2 \\ \texttt{i+1:} & x := 0 \end{array}$$

By enforcing the original control flow of the program, after taking any added instrumentation into account with dealing with instruction addresses, we prevented the attacker from bypassing our inlined safety policy enforcement. But notice that it didn't really matter what control flow graph we started out with. It could have been arbitrary, perhaps completely different from the actual CFG of the original program, and we could still enforce it by inserting and replacing conditional jumps.

4 Security automata

The point mentioned at the end of the previous section raises some interesting possibilities. What if we want to enforce a more general safety property that takes aspects of control flow and program state into account? For example, suppose that our language from the previous two sections has the ability to make three types of system calls, `send` and `recv` from the network and `read` from a local file. Then we quite naturally might want to enforce a safety policy on untrusted code which says that `send` cannot be called after `read`.

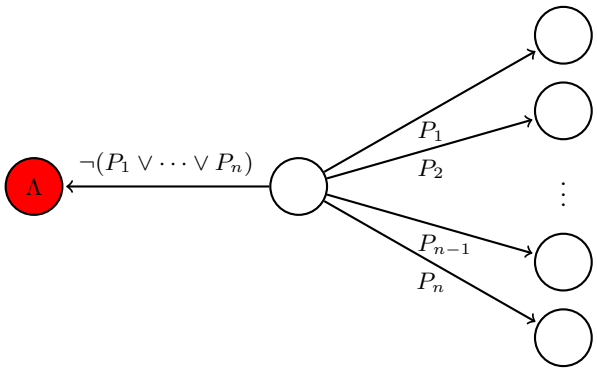
We can encode such a policy using a *security automaton* [2]. Depicted below is a security automaton for the safety policy "no send after read". The states are abstract in the sense that they do not reflect anything about the state of the program or what it is currently doing. Rather, they represent the state at which the policy is currently in. The transitions reflect facts about program state that must be true in order for the automaton to transition. In this case, `pc` denotes the current program counter, and `*pc` its contents. So for example `*pc` $\neq$ `read` corresponds to states in which the current instruction pointed to by the program counter is not a network read `read`.



Notice that the only arrow going out of the rightmost state is a self-loop labeled `*pc` $\neq$ `send`. There are no accepting states in a security automaton, and the way to interpret them is that as long as the automaton can transition from some arrow in its current state, then the policy has not been violated. So in this case, if the current policy state were the rightmost one, and the program entered into a state where `*pc` $=$ `send`, then there would be no arrow to transition from and the policy would become violated.

Another way to think about it is that there is a "hidden" error state which corresponds to the policy being violated. Every node has a transition to the error state on

the condition that is the negation of all other outgoing transitions from that state, as shown in the diagram below.



These definitions are equivalent, and we will continue using the convention that does not explicitly list the error state as this will reduce clutter in our diagrams.

4.1 Enforcing security automata policies

The primary means of enforcing policies defined using security automata is with a *reference monitor* (RM). The RM is a mechanism that examines the program as it executes, using information about the current and past states to decide whether the policy has been violated. This is done according to Definition 3, and was sketched out at the beginning of this section.

Definition 3 (Security automaton enforcement)**.** Let S be the current set of states that the security automaton is in, and $\delta(S, \omega)$ be a transition function that specifies a new set of states to enter given S and program state ω. Then for each step that the program is about to take resulting in new state ω, the reference monitor does one of two things.

1. If the automaton can transition, i.e. $\delta(S, \omega) \neq \emptyset$, then the program is allowed to enter state ω and the automaton updates its current states to $\delta(S, \omega)$.
2. If the automaton cannot transition, i.e. $\delta(S, \omega) = \emptyset$, then the program is not allowed to enter state ω and the reference monitor takes remedial action.

As long as the policy is not violated, then the RM allows the program to continue executing as it otherwise would. If the policy is violated, then the RM intervenes on the program execution to take some remedial action. This could mean simply aborting the execution, or something less drastic that prevents harm in other ways.

Necessary assumptions. As pointed out by Schneider in his seminal work on security automata [2], there are several assumptions that one must make in order to enforce these policies effectively with a reference monitor. First, the reference monitor needs to simulate the execution of the automaton as the program runs, so it must keep track of which state the policy is in on the actual hardware running the program. This means

that the automaton cannot require an unbounded amount of memory, so automata that have an infinite number of states are not in general enforceable.

Second, the RM must be able to prevent the program from entering a state that would result in a policy violation. This is called *target control*, and is a more subtle issue that it may at first seem. Take for example the policy of "real-time" availability, which states that a principal should not be denied a resource for more than n real-time seconds. How could a reference monitor enforce this policy? It might try to predict the amount of time that it takes to remediate a trace that is about to violate the policy, and take action earlier than necessary to prevent the violation. But how does it know that the policy would have actually been violated in this case? Unless the reference monitor can literally stop time, this is not an enforceable policy.

Third, the program under enforcement must not be able to intervene directly on the state of the reference monitor. This is called *enforcement mechanism integrity*, and is crucial for ensuring that the policy defined by the automaton is the one that is actually enforced on the target program. We dealt with an instance of this issue earlier in the lecture, when we used control flow integrity to make sure that inlined safety checks weren't bypassed by indirect jumps. But now that the policy itself has state, the enforcement mechansim must also guarantee that the target program does not make changes to that state or influence it in any way that doesn't follow the automaton transitions.

Inline SA enforcement. One approach to implementing security automata enforcement uses inlined checks to update and maintain state set aside to simulate the automaton. If we assume that formulas on SA transitions are formulas over program states, and there are N security automata states, then we can set aside a region of N memory cells at addresses a_{sa} through $a_{sa} + N$ to hold the current state of the automaton. If $\texttt{Mem}(a_{sa} + i)$ is non-zero, then we assume that the automaton has entered into state i, and otherwise not.

Next we need to implement the transition function, updating the contents of $\texttt{Mem}(a_{sa})$–$\texttt{Mem}(a_{sa} + N)$ to simulate the automaton. Suppose that the automaton has an edge from states i to j labeled with formula P. Then for each instruction in the program we compute the verification condition of (5).

$$[\alpha]P \tag{5}$$

If (5) is satisfiable before executing α, then it means that there may be a trace where P is true after executing α. This means that we need to insert instrumentation immediately before α that checks $\texttt{Mem}(a_{sa}+i) \neq 0 \wedge [\alpha]P$, and if it is true then sets $\texttt{Mem}(a_{sa+j})$ to a non-zero value. Then for each state i in the SA, we compute similar checks for transition to the "error state". If i has outgoing edges labeled $P_1, \ldots, P_n$, we insert a check for:

$$\texttt{Mem}(a_{sa} + i) \neq 0 \wedge [\alpha]\neg(P_1 \vee \cdots \vee P_n) \tag{6}$$

If this check passes, it means that the automaton cannot transition from state i. If this holds for every state in the automaton, then the instrumentation aborts execution.

The instrumentation described so far only addresses updates to the SA state. We must also take steps to ensure the integrity of the inlined mechanism, and there are two sources of vulnerability.

- The contents of $\texttt{Mem}(a_{sa}) - \texttt{Mem}(a_{sa} + N)$ must not be modified by any part of the program except the inserted instrumentation. Applying software fault isolation to the untrusted instructions can ensure that this aspect of integrity holds.
- The inserted instrumentation could be subverted by indirect control flow. Enforcing CFI on the untrusted code using the original control flow graph ensures that this will not happen.

This is sufficient to implement a basic inlined security automaton enforcement mechanism. However, it may impose a severe performance overhead due to all the safety checks. There are ways of mitigating this downside that rely on static analysis and techniques from compiler optimization, but they are outside the scope of this class.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. *ACM Transactions on Information and Systems Security*, 13(1):4:1–4:40, Nov. 2009.

[2] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information Systems Secur.*, 3(1):30–50, Feb. 2000.

[3] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary cpu architectures. In *Proceedings of the 19th USENIX Conference on Security*, 2010.

[4] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of CCS 2007*, Oct. 2007.

[5] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy*, 2009.