

# Lecture Notes on Security Automata

Software Foundations of Security and Privacy  
Matt Fredrikson

## Lecture 10

### 1 Introduction & Recap

We began studying safety properties by intuiting that they describe systems where something “bad” never happens, and have seen that contracts, assertions, memory safety, and more granular forms of sandboxing are all instances of safety. But there are certainly other types of bad events that we might want to write policies to enforce against, and aside from finding a way to encode them in programs using assertions, it isn’t clear how we would go about doing this.

In Lecture 8 we developed symbolic evaluation, an algorithmic proof procedure for dynamic logic judgments about programs. Symbolic evaluation works with sequents of the form

$$\Gamma \Vdash S$$

where  $\Gamma$  is a set of pure assumptions (path constraints) and  $S$  is a stack built from pure formulas and box modalities (e.g.,  $Q$  or  $[\alpha]S$ ). The proof rules mirror evaluation order: composition pushes a continuation onto the stack, assignments introduce fresh variables/equalities, and conditionals split into branches while extending  $\Gamma$ . When the succedent is pure, we discharge the goal using an arithmetic oracle.

One useful way to read symbolic evaluation is as a symbolic execution tree. Each branch corresponds to a family of concrete executions, and the evolving assumptions  $\Gamma$  describe the concrete program states that can arise along that branch.

Today we shift from proving properties about programs to enforcing policies by monitoring a program as it executes. A runtime monitor observes the program’s execution trace (states, events, or other chosen observations), maintains its own internal state, and intervenes when it detects that execution is about to violate a policy.

This raises a fundamental question: which policies can be enforced by watching a program execute and intervening based only on what we have observed so far? In this lecture we study how to characterize this enforceable class and how to represent such policies using security automata [2], small state machines that read the observed trace and reject as soon as a bad prefix appears.

## 2 Security automata

When a program runs, it produces an observable *trace* that we can think of as a sequence

$$\sigma = \sigma_0, \sigma_1, \dots$$

of “steps” of execution. In the most concrete view, a step might be the entire program state  $\omega$  (a valuation of variables, memory, etc.). But for many policies we instead view each step as an *event*: a system call, an API invocation, a message received over the network, and so on. The point is that a safety policy rules out certain *bad prefixes* of this trace.

A *security automaton* is a convenient way to represent and enforce such policies: the automaton’s state summarizes the relevant history of the trace so far, and each transition is guarded by a predicate describing which next steps are allowed. We will use the convention that we draw only the transitions that keep the policy satisfied; if no transition is enabled, the current prefix must be bad and the policy is violated.

### 2.1 Runtime-enforceable trace properties

At runtime, an enforcement mechanism only observes a growing *prefix* of the trace and can react by intervening (e.g., aborting execution). This imposes some basic structure on the kinds of policies we can hope to enforce.

- **Trace condition.** A policy must be a condition on a program trace. That is, whether the policy holds should be determined only by the observed sequence  $\sigma$  of states/events that execution produces, as opposed to requiring information about other executions that did not occur, or external factors that may not be accessible when the program executes.
- **Prefix closure.** Once the policy is violated, later execution cannot “repair” it. Formally, if a prefix  $\tau$  is already a violating prefix, then any extension of  $\tau$  is also violating.
- **Finite refutability.** Violations must have a finite witness: if a full trace violates the policy, then there exists a finite bad prefix after which all extensions still violate. This is exactly what makes runtime detection possible: a monitor only needs to watch until a bad prefix appears.

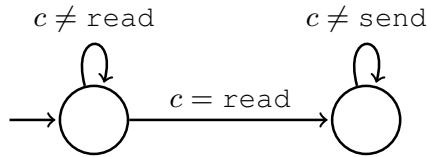
These are the hallmarks of *safety* properties: “nothing bad ever happens” means that if something bad happens, we can point to the finite moment when it first did.

### 2.2 Example: No send after readfile

Suppose our trace consists of system-call instructions/events such as `read` (read from a local file) and `send` (send to the network). We want to enforce:

after a `readfile`, there must not be any subsequent `send`.

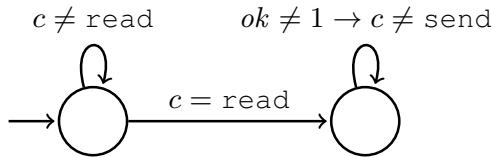
Let  $c$  denote the system call executed at the current step. The following security automaton enforces this policy.



There are no accepting states in a security automaton: as long as there is some enabled transition from the current policy state, the policy has not been violated. Here, once we have seen a `read` event, we move to the rightmost state, where only events satisfying  $c \neq \text{send}$  are permitted. If the program executes `send` at that point, there is no transition to take and the monitor concludes that the prefix is bad.

Sometimes we care about both which event occurred and some property of the program state at that moment. One simple way to model this is to let the input symbol include both pieces of information (e.g.,  $\sigma_i$  could be a pair consisting of the current state and the current event), and then write transition guards that mention both.

For example, we might relax the previous policy to allow a `send` after `read` only if a program variable `ok` indicates the data has been sanitized. When  $ok = 1$  the implication is trivially true (so `send` is permitted), and when  $ok \neq 1$  it reduces to  $c \neq \text{send}$ .



If the program attempts to execute `send` when  $ok \neq 1$ , then the guard is false and there is no enabled transition, so the policy is violated.

## 2.3 Definition

Next we define security automata and what it means for them to accept traces.

**Definition 1 (Security automaton[2])** *A security automaton is a nondeterministic state machine that consists of the following components:*

- a countable set  $O$  of automaton states,
- a countable set  $O_0 \subseteq O$  of initial states,
- a countable set  $\Sigma$  of transition symbols (the alphabet of observable steps/events in traces  $\sigma$ ),
- a transition relation  $\delta \subseteq O \times \wp(\Sigma) \times O$  between automaton states and sets of transition symbols.

We will assume that sets of transition symbols are represented by formulas (predicates) that can be evaluated over input symbols (events). Concretely, the set of symbols corresponding to a formula is comprised of exactly the symbols that the formula is true of. Given a sequence of transition symbols (events)  $\sigma = \sigma_0, \sigma_1, \dots$ , we say that the automaton accepts  $\sigma$  if and only if there is a corresponding sequence of states  $o = o_0, o_1, \dots$  such that for each pair  $\sigma_i, \sigma_{i+1}$  in  $\sigma$ ,

- there is a corresponding pair  $o_i, o_{i+1}$  of states in  $o$ ,
- and there exists  $(o_i, P, o_{i+1}) \in \delta$  where  $\sigma_i \models P$ .

In other words, a trace is only accepted if there is a corresponding run of the automaton that always follows the transition function.

The choice of transition symbols  $\Sigma$  is a design decision: it defines what the monitor gets to observe. In some settings a symbol might be a system call event like `read` or `send`, as in our examples above. In others, a symbol might be a full program state, or a structured observation such as a pair consisting of an event together with a snapshot of selected state. The key requirement is that the symbol stream contains enough information to recognize bad prefixes of the safety policy.

## 2.4 Taint analysis

We have seen that information flow cannot be checked by examining single traces in isolation. But this doesn't mean that they can't be *approximated* in some useful sense by safety properties. One approach for approximating information flow that has been used widely for certain applications is called *taint analysis* [3].

Conceptually, taint analysis enforces a safety property that tracks the portions of program state that have been "tainted" by some identified source. The policy maintains state that contains a bit for each variable and memory cell used by the target program. A subset of the variables and memory cells are distinguished as taint "sources", and a different subset as the "sinks". Then as the program executes, the policy state is updated to reflect the flow of tainted information through the state. If any of the state identified as a sink becomes tainted, then the policy is violated.

**Security automaton.** Taint analysis can be formalized and enforced as a security automaton. We show how by defining the states, initial states, transition symbols, and transition relation. To keep things simple while illustrating the main ideas, we will assume that the language has only variables and no memory state or operations.

- The states of the automaton correspond to the set of all *taint mappings*  $T$  from program variables to  $\{0, 1\}$ . Intuitively, if the policy is in a state where  $T(x) = 1$ , then  $x$  is currently tainted, and if  $T(x) = 0$  then it is not.
- The initial states are all mappings  $T$  where  $T(x) = 1$  for each identified source variable and  $T(y) = 0$  for all non-source variables.

- The transition symbols are program instructions  $x := e$ ,  $\text{assert}(Q)$ ,  $\text{if}(Q)$  jump  $e$ .
- The transition relation is defined to *propagate* the taintedness of an expression on the right-hand side of an assignment to the variable on its left-hand side. If  $T$  is a taint mapping, then we use the following rules to determine whether an expression is tainted.

$$(\text{Var}) \quad \frac{T(x) = 1}{T \vdash x} \quad (\text{OpL}) \quad \frac{T \vdash e}{T \vdash e \cdot \tilde{e}} \quad (\text{OpR}) \quad \frac{T \vdash \tilde{e}}{T \vdash e \cdot \tilde{e}}$$

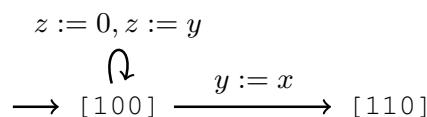
Then for every assignment instruction  $x := e$  where  $x$  is not a sink variable, and pair of states  $T_1, T_2$  where  $T_1 \vdash e$  and  $T_2(x) = 1$ , the corresponding edge  $(T_1, x := e, T_2)$  is in the transition relation. Additionally, for every pair of states  $T_1, T_2$  where  $T_1 \not\vdash e$  and  $T_2(x) = 0$ ,  $(T_1, x := e, T_2)$  is in the relation. Finally, for every other instruction  $\alpha$  that is not an assignment and every state  $T$ ,  $(T, \alpha, T)$  is in the transition relation.

Let's see how this works for a simple program with variables  $x, y$ , and  $z$ . Suppose that  $x$  is a taint source, and  $z$  is the sink:

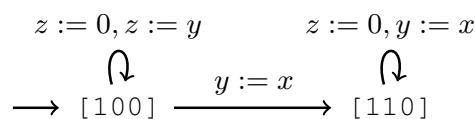
$$\begin{aligned} 1 : & z := 0 \\ 2 : & y := x \\ 3 : & z := y \end{aligned} \tag{1}$$

The automaton will have eight states, as there are  $2^3 = 8$  mappings from the three variables to  $\{0, 1\}$ . We will use a shorthand to denote states that gives the value of the mapping on each variable  $x, y, z$  in order. So the initial state maps  $x$  to 1, and  $y, z$  to 0 and is denoted  $[100]$ .

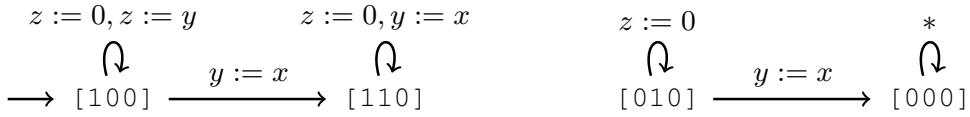
We will only consider edges corresponding to instructions in the program, so the initial state has three possible outgoing edges. On executing either  $z := 0$  or  $z := y$ , none of the taint state changes so we would remain in the initial state. On executing  $y := x$  however, the mapping for  $y$  changes to 1 because  $[100] \vdash x$ . So the automaton can transition to  $[110]$ .



From state  $[110]$ , executing  $z := 0$  or  $y := x$  leaves the taint mapping the same, but executing  $z := y$  would result in  $[111]$ . However,  $z$  is a sink variable so entering this state would be a policy violation, and we leave the edge out.



The full automaton has additional states and transitions. For example, if the taint mapping were  $[010]$ , then executing  $y := x$  would transition to  $[000]$ , whereas executing  $z := 0$  would stay in the same state. Moreover, executing any instruction in  $[000]$  stays in that state, so we would add this to the automaton.



But these states are not reachable from the initial state of the SA, so we can leave them out and stop here.

**Implicit flows.** So far taint analysis seems to do what we want in terms of tracking information flow as the program executes. So why did we say that it is an approximation of information flow, and moreover, why doesn't this contradict our thought experiment from earlier?

Recall that the program  $\alpha_1$  in (??) flowed information from  $x$  to  $y$ , effectively computing  $y = (1 - x)$ . But it did not compute this using an assignment with  $x$  on the right hand side, and rather with a conditional statement and constant assignments in either branch. This is known as an *implicit flow* because the information moves from  $x$  to  $y$  indirectly by the choice of which program to execute within the conditional, rather than explicitly via an assignment statement.

The taint analysis policy that we described doesn't deal with implicit flows because it only propagates taintedness through explicit assignments. In this way taint analysis *underapproximates* information flow. Whenever taint analysis deems that information has flowed from a source to a sink, then an information flow exists in the program. But if it fails to identify a flow, then we can't conclude that there isn't one in the program because there could yet be an implicit flow.

Can we change the security automaton policy to account for implicit flows? Suppose that we added an additional mapping to the policy state, intuitively corresponding to whether the program counter is tainted. We'll denote this  $T(\text{pc})$ , and we want to make sure that it is set to 1 whenever a conditional statement on a tainted expression is executed. So for example, if  $x$  is a source in program (2) below, then the policy will be in a state where  $T(\text{pc}) = 1$  after executing the instruction at 1.

1 : if( $x \neq 0$ ) jump 4 2 : $y := 0$ 3 : if(true) jump 5 4 : $y := 1$ 5 : ...	(2)
---	-----

Then we can adapt the policy so that whenever  $T(\text{pc}) = 1$  in the current state, the target of *any* assignment will become tainted. In (2), this will ensure that  $y$  is tainted after the conditional at 1, and thus reflect the fact that there is an information flow from  $x$  to  $y$  in the program.

But now consider what happens if we add another instruction, as in (3).

```

1 : if( $x \neq 0$ ) jump 3
2 :  $y := 0$ 
3 : if(true) jump 5
4 :  $y := 1$ 
5 :  $z := 0$ 

```

(3)

Now under the current policy,  $z$  will become tainted because it is executed after the tainted conditional. But there is clearly no flow of information from  $x$  to  $z$ , so the taint mapping no longer reflects the actual flows in the program. Perhaps we could refine the policy further, by un-tainting the program counter once control returns to instructions that are in no way conditional on tainted data. In (3), this is true for the assignment to  $z$  at instruction 5 because it is executed regardless of the condition at 1.

But how do we identify instructions that aren't conditional on tainted data in general? It was easy in this program, but things could get much more complicated as shown in (4).

```

1 : if( $x \neq 0$ ) jump 6
2 :  $y := 0$ 
3 : if(recv()  $\neq 0$ ) jump 7
4 : if(true) jump 6
5 :  $y := 1$ 
6 :  $z := 0$ 
7 : ...

```

(4)

Now in order to tell whether to untaint the program counter before executing instruction 6, we need to predict the outcome of a network read. This policy has gotten significantly more complicated, and keep in mind that we need to do all of these calculations at runtime, which may impose considerable overhead on the target program.

Finally, even if we could solve this problem, tracking taint across implicit flows still may not accurately reflect a program's information flows. Consider the program in (5) below.

```

1 : if( $x \neq 0$ ) jump 3
2 :  $y := 0$ 
3 : if(true) jump 5
4 :  $y := \text{complicated\_function}()$ 
5 : ...

```

(5)

It may turn out that `complicated_function()` always returns 0, so that  $y$  always takes the same value after the conditional regardless of what  $x$  is. In this case, there is no real flow of information from  $x$  to  $y$ . But our security automaton policy can't reason this way, because it would require considering a different trace from the one that was actually executed. Even if it could, it would need to reason about the behavior of an arbitrarily complicated, possibly non-terminating function, which is clearly not possible.

## 2.5 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 2, and was sketched out at the beginning of this section.

**Definition 2 (Security automaton enforcement)** *Let  $O_c$  be the current set of states that the security automaton is in. Then for each step that the program is about to take resulting in new program state  $\omega$ , the reference monitor does one of two things.*

1. *For each state  $o \in O_c$  that the automaton can transition from, the states  $\delta(o, P, o')$  for all transition edges where  $\omega \models P$  are added to the new automaton states.*
2. *If the automaton cannot transition from any of its current states, then the program is not allowed to enter state  $\omega$  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 than 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 mechanism 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.** A common way to implement a reference monitor is to inline it: we rewrite the program to include additional checks and updates that simulate the security automaton as the program runs.

At a high level, we keep some extra state (in memory or registers) that represents the current automaton state(s). Before each instruction executes, we (1) decide which automaton transitions are enabled next, and (2) update the stored automaton state accordingly. If no transition is enabled, we take remedial action (e.g., abort).

When transition labels are predicates over program variables, weakest precondition gives a direct recipe for what to check before executing an instruction. If a transition into some next policy state requires that a predicate  $G$  holds after executing instruction  $\alpha$ , then it suffices to check the weakest precondition  $\text{wp}(\alpha, G)$  in the current state.

## 3 Dynamic instrumentation

We have been discussing policy enforcement in a somewhat idealized model, where we assume that programs are given to us as source code in a simple language with few instructions. In the "real world" this is not usually the case, and we may be forced to deal with large untrusted programs given to us to execute at runtime, and possibly without source code. So we must find a way to enforce policies on bytecode, and presumably fast lest we introduce unacceptable latency into the system.

Suppose that we wish to implement the inline security automata enforcement scheme from the previous section by changing the instructions throughout the program prior to running it. This seems like a reasonable approach, because the scheme just requires that we check verification conditions on each instruction and replace them when necessary. All that we need to assume is the ability to identify instructions, and compute verification conditions.

### 3.1 Challenges for static instrumentation

But bytecode programs on modern architectures like x86 and AMD64/Intel 64 are extremely difficult to reason about statically, and it may not even be possible to identify which instructions the program will end up executing. One practical issue is the fact that programs can generate new instructions by writing to memory, and then use an indirect jump to begin executing the newly-written code. This can be mitigated by the operating system with a *Write XOR Execute* policy, which ensures that any page of memory may be either writeable or executable, but not both. This is effective, but makes some functionality extremely difficult to implement such as language interpreters that do on-the-fly compilation and optimization.

Even with Write XOR Execute, the presence of indirect control flow and variable-length instruction encoding makes it impossible to tell which instructions will actually be executed. The program can do an arbitrarily complicated computation to derive a target address in existing code, so that the static analysis is unable to determine where execution will resume after a jump. If the target address is in the middle of an existing instruction, it may result in a completely different program being executed. Consider the following example, taken from [4].

<i>Bytecode</i>	<i>Instruction</i>	
f7 c7 07 00 00 00	test \$0x00000007, %edi	(6)
0f 95 45 c3	setnz -61(%ebp)	

This code is taken from the entry point of an encryption routine in the GNU C library, often referred to as simply libc. If execution begins one byte after the entry point of (6), a completely different program is executed.

<i>Bytecode</i>	<i>Instruction</i>	
c7 07 00 00 00 0f	movl \$0x0f000000, (%edi)	
95	xchg %ebp, %eax	(7)
45	inc %ebp	
c3	ret	

Importantly this implies that given a sequence of bytecodes, there are numerous possible programs that could end up being executed depending on which addresses are targeted by indirect jumps. In order to instrument the right one, a static analysis needs to determine what these addresses will be, and this is an undecidable problem in general. Moreover, it could be that information not available statically, such as network packets, are used in part to compute target addresses, adding yet another very plausible complication for static instrumentation in this setting.

### 3.2 Instrumenting with just-in-time compilation

Perhaps a better approach given these challenges is to delay “code discovery” until the program is actually running. This is helpful for many reasons.

- If the program generated instructions in memory and transferred control to them, we no longer need to infer what those instructions will be. We can simply wait until the program has already written them, and instrument them immediately before the control transfer.
- If a program executes an indirect jump, we do not need to predict what the target address will be. We simply wait until immediately before the jump is executed, at which point the target address will be stored in memory or a register, and begin instrumenting the target of the jump.

- Some other cases that we have not discussed are handled similarly, such as libraries that are loaded after the program begins executing. In each such case, the instrumentation is delayed until immediately before the instructions in question begin executing, at which point all of the necessary information is available.

The obvious drawback to this approach is the fact that we need to examine the execution as it unfolds, rewriting instructions whenever necessary as dictated by the policy.

**Just-in-time compilation.** A successful and widely-deployed approach to mitigate the performance penalty imposed by such a scheme is called *just-in-time (JIT) compilation* [1]. The key insight behind JIT compilation is to increase the granularity at which the instrumenter examines code at runtime, looking at “chunks” of instructions rather than individual ones.

Increasing the granularity in this way allows the instrumenter to compile instruction chunks, with their instrumentation included, on the fly into optimized code that is then executed directly. Further performance enhancements can then be layered on top of this basic approach, such caching previously-compiled chunks to save redundant work, as well as more aggressive optimizations to sequences of chunks that end up being executed more often.

The question then becomes what constitutes a chunk. Larger chunks will generally create more opportunities for optimization, and because more of the instructions are dealt with each time, require fewer (expensive) calls to the compiler. However, this tendency is limited by the fact that if a chunk crosses an indirect control flow instruction, then we run into exactly the same problems we are trying to avoid with dynamic instrumentation in the first place. Even if our chunks cross direct, predictable control flow branches, then we run the risk of doing unnecessary compilation and instrumentation by processing multiple branches when the execution will only end up following one of them.

The typical approach is to use *basic blocks* as chunks. A basic block is a contiguous sequence of instructions that ends in a control flow transfer instruction (e.g., `jmp`, `ret`, `call`, ...). For example, the sequence of instructions in (7) is a basic block because it ends with a `ret` instruction, which transfers control to the instruction pointed to by the return address on the stack. On the other hand, (6) is not a basic block because it does not end in such an instruction.

Using basic blocks as chunks, the instrumenter will begin scanning a sequence of bytecodes until it reaches a control transfer instruction. It will then instrument each of the instructions in the basic block as prescribed by the policy, compile the resulting instructions, and execute them. However, it must ensure that it regains control when the basic block is finished executing. It then begins scanning instructions again at the bytecodes pointed to by the instruction pointer, repeating the process all over again. In this way we can be sure that exactly the code that is executed is instrumented according to the policy.

## References

- [1] John Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, June 2003.
- [2] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information Systems Secur.*, 3(1):30–50, February 2000.
- [3] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (Oakland)*, 2010.
- [4] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of CCS 2007*, October 2007.