15-316: Software Foundations of Security and Privacy

Lecture Notes on Security Automata & Instrumentation

Matt Fredrikson

Carnegie Mellon University
Lecture 11

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.

Today we will generalize what we have learned about enforcing safety by first encoding the bad prefixes using automata, and then monitoring a trace as it evolves though exection. This style of policy, called *security automata* [Sch00], is powerful enough to encapsulate all of the safety properties that can possibly be enforced at runtime, and is thus an indespensible tool for ensuring code safety.

2 Security automata

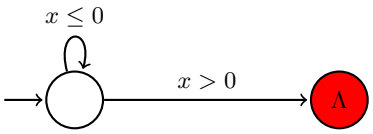
To begin developing an intuition for how safety properties can be represented using automata, we return to invariant properties—arguably the simplest type of safety property that we studied. Suppose that we want to enforce a condition on x, e.g.,

$$0 \leq x$$

In terms of bad prefixes, we could characterize this as any trace containing a state in which $x < 0$. We could imagine "watching" the trace as it develops, and if it ever enters such a state, we would know that safey has been violated. This type of monitoring can be formalized by constructing a finite automaton, whose states represent the status of

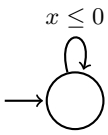
our policy, and whose transitions correspond to the current, most-recent state of the program's trace.

In this case, our simple policy has a single status, namely whether $0 \leq x$ has remained true so far in the trace. While we may observe many distinct states as the program executes, the only fact that matters from our perspective is whether this predicate remains true in the current state. To capture all this, we can draw the following automaton.



This tells us everything that we need to know to enforce the invariant as we watch the trace unfold. We begin in a state where the policy has not yet been violated, because the program has not done anything yet. At each moment before a command is executed, we examine the current state, and match one of the conditions on the transitions to update the status of the policy. If we ever reach the error state, then we must have seen a bad prefix, and the policy is violated.

Notice that we do not really need to make the error state quite so explicit, and could just as easily leave it implicit.



Here, rather than waiting to enter the error state, we merely want to make sure that there always exists some transition with a label that describes the most recent state in the trace. If we ever see a state that does not satisfy any of the transition labels emanating from the current policy state (i.e., one where $x > 0$), then we conclude that the current prefix must be bad, and the policy is violated. This is called a *security automaton* [Sch00], and is the main abstraction that we will use to encode safety in this lecture.

2.1 A more interesting example.

As the states comprising traces in our language consist of variable and memory mappings, we have so far been unable to formalize properties that account for the commands that are executed over time. For example, if our language has the ability to make system calls, such as `send` and `recv` from the network and `read` from a local file, we might want to enforce a policy which says that a program should not send data over the network after it has read from local files. This is indeed a safety property, but to understand it in terms of bad prefixes, we would need to extend the states in our semantics to reflect the commands that are executed.

Accounting for commands. Namely, we could formalize states as triples $(\omega_V, \omega_M, \texttt{*pc})$ consisting of variable mappings ω_V, memories ω_M, and "program counters" `*pc`. Unlike ω_V and ω_M, the `*pc` component is not a mapping, but rather a value ranging over programs. We could easily extend the semantics of the language to maintain this state, for example:

$$[\![x := e]\!] = \begin{array}{l} \{(\omega, \nu) \ : \ \omega[\![e]\!] \text{ defined}, \nu = \omega \text{ except } \nu_V(x) = \omega[\![e]\!], \ \nu_{\texttt{*pc}} = x := e\} \cup \\ \{(\omega, \Lambda) \ : \ \omega[\![e]\!] \text{ is not defined}, \ \nu_{\texttt{*pc}} = x := e\} \end{array}$$

There are multiple ways that we could update the semantics of compound commands like $\alpha; \beta$ and $\texttt{if}(P)\, \alpha\, \texttt{else}\, \beta$. The most straightforward would be to leave their semantics unchanged, merely carrying the updates to `*pc` made by the *atomic* commands forward. In other words, consider the following program.

$$x := y; \texttt{Mem}(x) := 0$$

The traces of this program have three states $\omega_0, \omega_1, \omega_2$. Because nothing has executed in the initial state, we would assume that `*pc` takes a "null" value, e.g., $\varnothing$. After the first command executes, `*pc` in ω_1 would be $x := y$, and in the final state, $\texttt{*pc} = \texttt{Mem}(x) := 0$. This would be a consequence of maintaining the usual semantics for sequential composition:

$$[\![\alpha; \beta]\!] = \{\sigma \circ \varsigma \ : \ \sigma \in [\![\alpha]\!], \varsigma \in [\![\beta]\!]\}$$

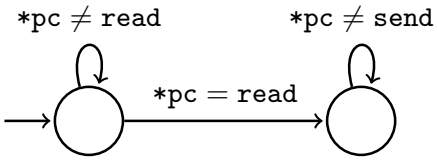
In other words, we want `*pc` to point to the most recent command that resulted in a new state, and the commands that cause this are atomic (i.e., variable update, memory update, and `assert`).

The alternative would be to add a state when the compound command begins executing, so that the way in which the program is composed is reflected in traces.

$$[\![\alpha; \beta]\!] = \{\omega \circ \sigma \circ \varsigma \ : \ \omega = \sigma_0 \text{ except that } \omega_{\texttt{*pc}} = \alpha; \beta, \ \sigma \in [\![\alpha]\!], \varsigma \in [\![\beta]\!]\}$$

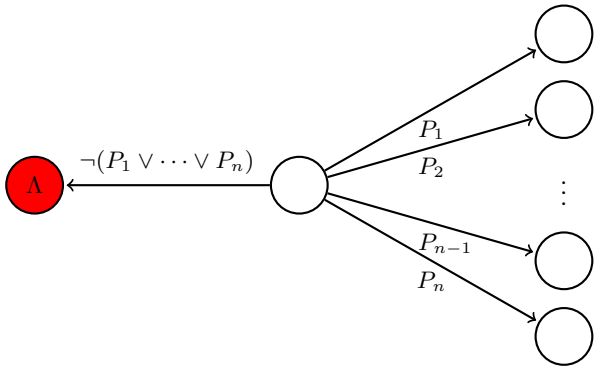
This approach might be useful if we need to define safety properties that depend on the composition of commands. However, the downside is that the updates to the semantics are more extensive, and slightly less intuitive. For the remainder of the lecture, we will assume that the semantics are updated only to record the execution of atomic commands, and the semantics of compound commands are left unchanged.

No send after read. 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 $\texttt{*pc} \neq \texttt{read}$ corresponds to states in which the current instruction pointed to by the program counter is not `read`.



Notice that as was the case with the invariant policy earlier, 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.

Definition 1 (Security automaton[Sch00])**.** 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 Σ of transition symbols,
- a transition relation $\delta \subseteq O \times \wp(\Sigma) \times O$ between automaton states and sets of transition symbols.

We will assume that the transition symbols always correspond to program states, and that sets of program states are represented by formulas that can be evaluated over program states. Concretely, the set of states corresponding to a formula is comprised of

exactly the states that the formula is true in. Given a sequence of transition symbols (i.e., states) $\sigma = \sigma_0, \sigma_2, \ldots$, we say that the autoamton accepts σ if and only if there is a corresponding sequence of states $o = o_0, o_1, \ldots$ such that for each pair σ_i, σ_{i+1} in σ,

- 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 $\sigma_i \vDash P$.

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

Definition 1 formally defines security automata in terms of a set of states O and transition symbols $\mathcal{S}$. We will generally assume that $\mathcal{S}$ is the set of all program states, so that we can describe program traces as being accepted or not by a security automaton. This also implies that sets of states in the transition relation are defined in terms of formulas on program states, which we have already studied extensively.

2.2 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 ω, 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 $\omega \vDash P$ 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 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 [Sch00], 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 (1).

$$[\alpha]\neg P \tag{1}$$

If (1) is valid before executing α, then it means that all traces after executing α will satisfy $\neg P$. On the other hand, if it is not valid, then at least one trace of α may satisfy P. This means that we need to insert a check whenever Eq 1 is not valid.

What check do we insert? At runtime, we will be in a particular state ω. We want to know if after executing α, P will be true, and if it is, then update the state of the automaton. We can accomplish this by simply checking that $\omega \models [\alpha]P$. Of course, we will want to use axioms to remove the box modality so that the check is actually in terms of arithmetic, and can be easily evaluated.

So we 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:

$$\texttt{Mem}(a_{sa} + i) \neq 0 \wedge [\alpha]\neg(P_1 \vee \cdots \vee P_n) \tag{2}$$

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.

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 [Sha07].

Bytecode	*Instruction*
`f7 c7 07 00 00 00`	`test $0x00000007, %edi`
`0f 95 45 c3`	`setnzb -61(%ebp)`

(3)

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 (3), a completely different program is executed.

Bytecode	*Instruction*
`c7 07 00 00 00 0f`	`movl $0x0f000000, (%edi)`
`95`	`xchg %ebp, %eax`
`45`	`inc %ebp`
`c3`	`ret`

(4)

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* [Ayc03]. 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 (4) 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, (3) 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.

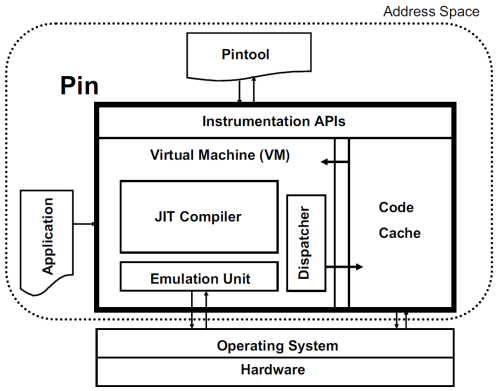


Figure 1: Pin software architecture (from [LCM+05]).

A look ahead: Pin. In your next lab, you will make use of an instrumentation tool called Pin [LCM+05] that is based on JIT compilation. Pin is under ongoing development by Intel, and is widely used in industry as well as in academic research. It simplifies the task of instrumenting binaries at runtime by providing a high-level API for both inspecting and instrumenting sequences of instructions at runtime.

To see why this is helpful, consider the task of instrumenting an x86 binary to prevent writes to certain portions of memory. To do so, we must rewrite all instructions that can change memory with instrumentation to stop the unwanted writes. Which instructions can change memory? The obvious ones are `mov`, `push`, `pop`, `lea`, `xchg`, and perhaps a few others. But what about the many variants of `mov`, such as `movsb`, `movsw`, `movz`, `movzx`? Do the other instructions have variants as well, and how can we be sure that we've covered each one? Pin simplifies things for us by providing `INS_IsMemoryWrite(ins)`, which returns true if `ins` can update memory.

Figure 1 shows the architecture of Pin. Users interact with it by writing a "Pintool", which is a conventional C or C++ program that makes use of the Pin inspection and instrumentation API. To run a compiled program under the pintool's instrumentation, the program's binary is passed to Pin along with the compiled pintool. Pin then takes care of just-in-time compiling the target program, and can invoke callbacks to the pintool as requested for inspection, or rewrite instructions as requested for instrumentation. As execution proceeds, Pin's optimization routines run in tandem to progressively optimize the compiled code.

You will learn more about the specifics of the Pin API in the handout for the next lab, and get hands-on experience using it to implement SFI as well as a security automaton policy. For more detailed information on how Pin works, consult the original paper [LCM+05].

References

[Ayc03] John Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, June 2003.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[Sch00] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information Systems Secur.*, 3(1):30–50, February 2000.

[Sha07] 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.