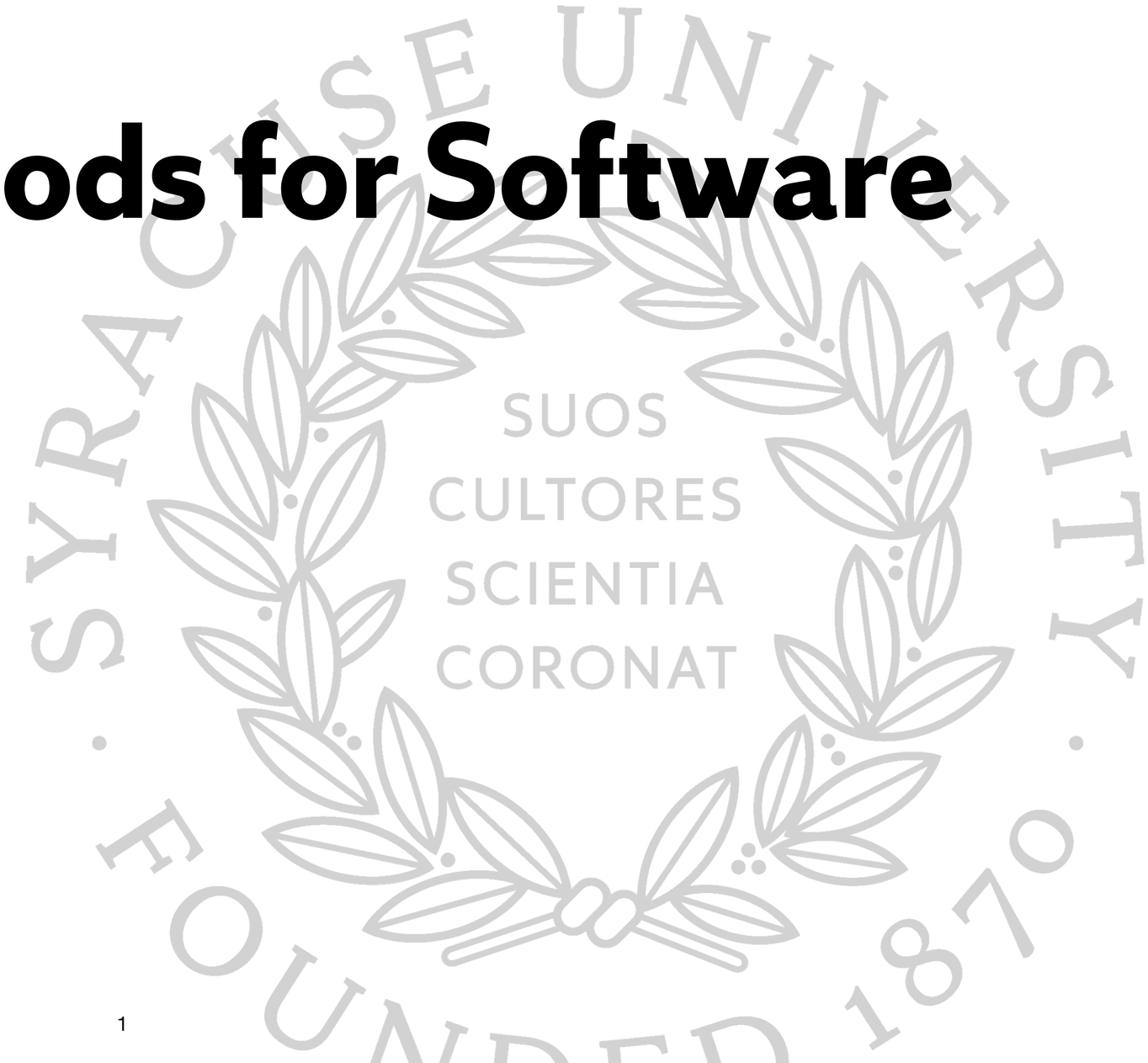
Formal Methods for Software Security

Kristopher Micinski

Syracuse University

For CySER, 2026



Software Security: Why Formal Methods?

- Many concepts from security are nuanced:
  - “What does it mean for a program to be secure?”
  - “Does the program leak any sensitive information?”

- The issue is that security is not a **functional correctness** property
  - Functional correctness: you can write a test case observing bug / no bug
  - Now even more useful (given generative AI) as agent input points

- **All** security statements only correct up to some attacker model
  - Formal methods: use mathematical logic to make statements about a program’s security, enables being precise

Example: is this app secure (in principle?)

- Webapp allows logins with username / password
- Failed login returns a user to the login page to reattempt
- All username & passwords <= N characters

Example: is this app secure (in principle?)

- Webapp allows logins with username / password
- Failed login returns a user to the login page to reattempt
- All username & passwords <= N characters

- This application is not secure in principle, why?
  - The app implicitly leaks whether or not the password is correct
  - Attack: just try all possible passwords
    - Practically **hard** (2^N) but not impossible in principle

How about this app, is it safe?

```
input0 = read_int() // read an integer from user
input1 = read_int() // read another
x = input0 + input1
y = x - x
output(y)
```

- Is the app secure now?
  - Answer: still depends on attack model, but since x-x=0 is an identity, the app always outputs y—this app is “non interfering:”
    - **Non-interference:** high-security ins don’t interfere with low-security outs
  - (Ultra aggressive attacker) What might we observe: (maybe?) differences in clock-cycle time depending on specific integers

Another example…

```
input0 = read_int()
input1 = read_int()
x = 0
if (input0 < input1):
  x = 1
output(x)
```

- Is the app secure now?
  - No: leaks the predicate, “first input less than second input?”
  - A predicate is a single binary (yes/no) decision
  - Sometimes we call a predicate a “bit,” because one bit of data is also a binary yes/no decision

What do formal methods say about these programs?

- Reasoning about security is clearly a bit tricky—we cannot even be precise without an attack model in mind, for instance
- This motivates applying formal methods—mathematically-rigorous, sound reasoning—to programs to help understand their security

- **Test** a program for a security exploit
- **Verify** the absence of any security exploits

- Which one of the above is more desirable?
  - Answer: Verify, since there are generally infinitely-many tests

- We will talk about several different approaches to formal methods for security
- **Program analysis** analyzes a program to approximate definite behavior
  - “We wrote this program, can we prove it secure?”
- **Security type systems** use type systems to rule out errors by construction
  - “We cannot write this program if it is not secure (type system rejects program)”
- **Program logics** allow us to calculate when a program is secure
  - “If we can solve these constraints, the program has an error—if not, no error”

What is the *behavior* of a program / system

- Need to model a program in *some* way…
- One common way: program…
  - *reads in an input stream* and *writes out an output stream*
- Here, the program's **semantics** is a *set of traces* of input/output streams

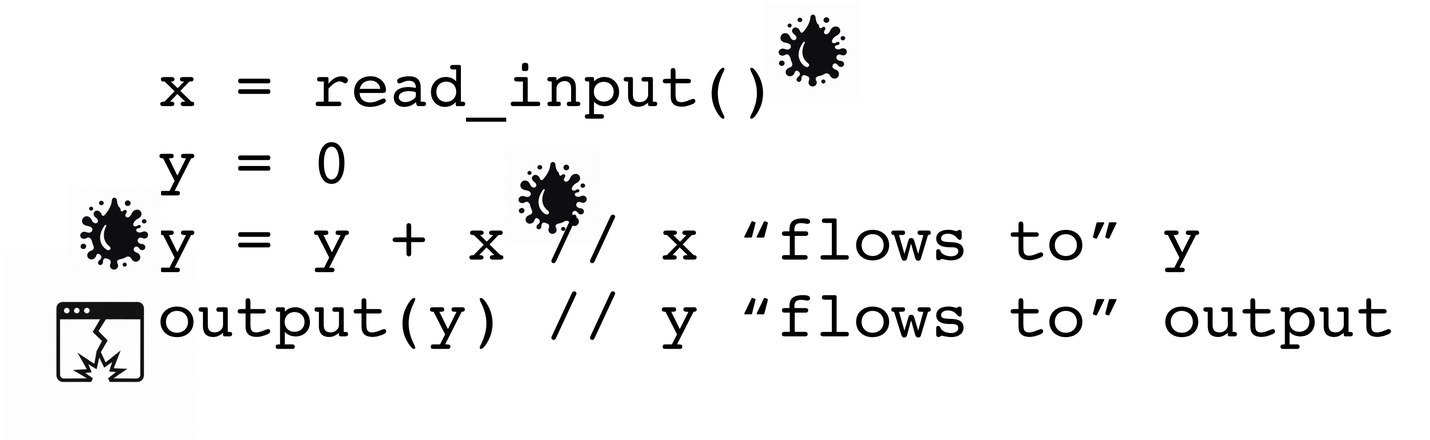
- out(0), out(1), out(2), out(3), …
- in(0), out(0), in(1), out(1), ….
- out(0)

Data Flow / Taint Tracking

- “No secret input **flows to** a low-security output”

```
x = read_input()
y = 0
y = y + x // x “flows to” y
output(y) // y “flows to” output
```

- We can check for this at **runtime** (“taint tracking”)
  - All inputs become “tainted”
  - Propagating a tainted input propagates “taintedness” (“infects” other values)
  - Program crashes if tainted value is written to output



Data Flow / Taint Tracking

- Taint tracking: either static or dynamic
- Dynamic taint tracking: propagate taint at runtime, (possibly) crash if there is a bad scenario
  - More like “testing,” you can observe bad behavior when it occurs
- Static taint tracking: analyze program at compile-time, ensure no possible misuse of data
  - More like “verification,” *prove* the absence of bad behavior

Static Program Analysis

- Static analysis is the computation of finite approximations of a program's behavior based on looking at (and computing over) its code, without running it

- Why is it bad to run the program?
  - *Can't* run the program in general—state-space explosion!
    - Fuzzing is great at exploring, but generally doesn't go "deep" into programs

- Thus, static analysis is a *requirement* if you want to make sound judgments!
  - *Necessary* for compiler optimizations—absolutely **cannot** break the program
  - Exceptions exist (tracing JITs, etc.)

Kildall ‘73

A UNIFIED APPROACH TO GLOBAL
PROGRAM OPTIMIZATION

Gary A. Kildall

Computer Science Group
Naval Postgraduate School
Monterey, California

Abstract

A technique is presented for global analysis of program structure in order to perform compile time optimization of object code generated for expressions. The global expression optimization presented includes constant propagation, common subexpression elimination, elimination of redundant register load operations, and live expression analysis. A general purpose program flow analysis algorithm is developed which depends upon the existence of an "optimizing function." The algorithm is defined formally using a directed graph model of program flow structure, and is shown to be correct. Several optimizing functions are defined which, when used in conjunction with the flow analysis algorithm, provide the various forms of code optimization. The flow analysis algorithm is sufficiently general that additional functions can easily be defined for other forms of global code optimization.

1. INTRODUCTION

A number of techniques have evolved for the compile-time analysis of program structure in order to locate redundant computations, perform constant computations, and reduce the number of store-load sequences between memory and high-speed registers. Some of these techniques provide analysis of only straight-line sequences of instructions [5,6,9,14,17,18,19,20,27,29,34,36,38,39,43,45,46], while others take the program branching structure into account [2,3,4,10,11,12,13,15,23,30,32,33,35]. The purpose here is to describe a single program flow analysis algorithm which extends all of these straight-line optimizing techniques to include branching structure. The algorithm is presented formally and is shown to be correct. Implementation of the flow analysis algorithm in a practical compiler is also discussed.

The methods used here are motivated in the section which follows.

2. CONSTANT PROPAGATION

A fairly simple case of program analysis and optimization occurs when constant computations are evaluated at compile-time. This process is referred to as "constant propagation," or "folding." Consider the following skeletal ALGOL 60 program: of the graph represent program control flow possibilities between the nodes at execution-time.

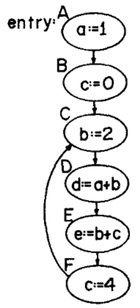


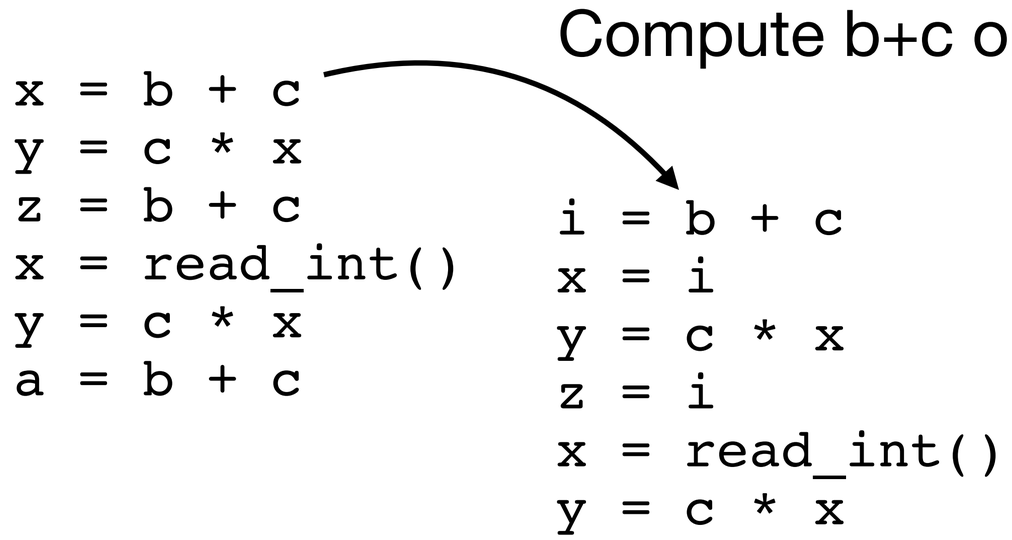
Figure 1. A program graph corresponding to an ALGOL 60 program containing one loop.

For purposes of constant propagation, it is

Available Expressions Analysis

- Dataflow analysis: a specific kind of static analysis which approximates how data flows through the program.

- Consider this example on the right…
  - b+c does not need recomputation
  - c * x needs recomputation
    - x's value changed



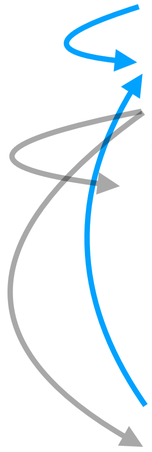
Available expressions analysis calculates:

"What expressions have *stayed the same* since they were last computed?"

Definition: Join Point (where multiple paths join together)

In this case, analysis facts ***flow forward*** unless "killed"

At *join points* we need to merge analysis results



```
1: t1 = b + c // GEN {b+c}
2: t2 = a + 1 // GEN {a+1}
header: // from line 2 {b+c, a+1}, from 8, {b+c}
3: if (n <= 0) goto end
body:
4: s  = b + c // GEN {b+c}
5: t3 = a + 1 // GEN {a+1}
6: a  = 0     // KILL exprs mentioning a (kills {a+1})
7: n  = n - 1 // KILL exprs mentioning n
8: goto header
end:
9: u = b + c
```

How do we merge two "sets of definitely-available expressions?"

Standard Intraprocedural Data Flow Analyses...

	May	**Must**
Forward	Reaching Definitions	Available Expresisons
Backward	Live Variables	Very Busy Expressions

And what they mean…

- Available expressions: which expressions have not changed since they were last computed — forward MUST analysis (must have been the same along all branches)
  - Common-subexpression elimination—compute a value ***once***
    - But what about 2 * x = x * 2? (Can't handle this entirely, not possible)
- Reaching definitions: which variables may reach a point without being overwritten
  - Dead code elim, constant propagation, computing use/def chains
- Live variables: which variables are demanded and unchanged since last assignment
  - Register allocation: non-live variables don't need to be in registers
- Very busy expressions: expressions which will necessarily be used along all paths
  - Loop invariant code motion—no need to recompute in the loop body

Cousot & Cousot ‘76

ABSTRACT INTERPRETATION : A UNIFIED LATTICE MODEL FOR STATIC ANALYSIS
OF PROGRAMS BY CONSTRUCTION OR APPROXIMATION OF FIXPOINTS

Patrick Cousot* and Radhia Cousot**

Laboratoire d'Informatique, U.S.M.G., BP. 53
38041 Grenoble cedex, France

1. *Introduction*

A program denotes computations in some universe of objects. Abstract interpretation of programs consists in using that denotation to describe computations in another universe of abstract objects, so that the results of abstract execution give some informations on the actual computations. An intuitive example (which we borrow from Sintzoff [72]) is the rule of signs. The text -1515 * 17 may be understood to denote computations on the abstract universe {(+), (-), (±)} where the semantics of arithmetic operators is defined by the rule of signs. The abstract execution -1515 * 17 ⟹ -(+) * (+) ⟹ (-) * (+) ⟹ (-), proves that -1515 * 17 is a negative number. Abstract interpretation is concerned by a particular underlying structure of the usual universe of computations (the sign, in our example). It gives a summary of some facets of the actual executions of a program. In general this summary is simple to obtain but inaccurate (e.g. -1515 + 17 ⟹ -(+) + (+) ⟹ (-) + (+) ⟹ (±)). Despite its fundamentally incomplete results abstract interpretation allows the programmer or the compiler to answer questions which do not need full knowledge of program executions or which tolerate an imprecise answer, (e.g. partial correctness proofs of programs ignoring the termination problems, type checking, program optimizations which are not carried in the absence of certainty about their feasibility, ...).

2. *Summary*

Section 3 describes the syntax and mathematical semantics of a simple flowchart language, Scott and Strachey[71]. This mathematical semantics is used in section 4 to built a more abstract model of the semantics of programs, in that it ignores the

Abstract program properties are modeled by a complete semilattice, Birkhoff[61]. Elementary program constructs are locally interpreted by order preserving functions which are used to associate a system of recursive equations with a program. The program global properties are then defined as one of the extreme fixpoints of that system, Tarski[55]. The abstraction process is defined in section 6. It is shown that the program properties obtained by an abstract interpretation of a program are consistent with those obtained by a more refined interpretation of that program. In particular, an abstract interpretation may be shown to be consistent with the formal semantics of the language. Levels of abstraction are formalized by showing that consistent abstract interpretations form a lattice (section 7). Section 8 gives a constructive definition of abstract properties of programs based on constructive definitions of fixpoints. It shows that various classical algorithms such as Kildall [73], Wegbreit[75] compute program properties as limits of finite Kleene[52]'s sequences. Section 9 introduces finite fixpoint approximation methods to be used when Kleene's sequences are infinite, Cousot[76]. They are shown to be consistent with the abstraction process. Practical examples illustrate the various sections. The conclusion points out that abstract interpretation of programs is a unified approach to apparently unrelated program analysis techniques.

3. *Syntax and Semantics of Programs*

We will use finite flowcharts as a language independent representation of programs.

3.1 *Syntax of a Program*

A program is built from a set "Nodes". Each node

Now, let's generalize…!

- Can compute ***any*** analysis by computing a *fixed point* of monotonic function
- ***Definition (Fixed Point of a Function):***
  - A fixed point of a function is a point X such that f(X) = X

- **Key issue**: need to know type of X for this statement to have any meaning!
  - Intuitively, X is FlowSolution (AnalysisResult, etc…)

- For the case of bit-vector dataflow analyses, X represents:
  - All of the In/Out sets for each statement in the program
  - (Avil. Expr.) X is finite map from statement (in/out) to set of AEs
- X (the type) must posses a notion of "no information" (written as ⊥)
  - For AE ⊥ is this map: "for every statement X, the In/Out set are both {}"

How do we make this rigorous?

- I just described one way of computing analysis solutions
  - The method I presented is called “chaotic iteration”
  - Claim: the process ***always terminates*** and ***always yields a fixed point***

- ***But***: How do we make this claim rigorous? Need a bit more formalism…

- We will show…
  - Monotonic functions always have a fixed-point
  - Which can always be computed via chaotic iteration!
  - First must define what a “monotonic” function is…

Definition: Lattice

- A *lattice* is a set E of lattice elements, along with…
  - A partial order $\sqsubseteq$, which relates elements ($e_0 \sqsubseteq e_1$) or not
    - Reflexive, antisymmetric, and transitive
  - An operation $e_0 \sqcup e_1$, the "join" (least upper bound) of e_0 and e_1
    - The least upper bound: the least element e′ s.t., $e_0 \sqsubseteq e'$ and $e_1 \sqsubseteq e'$
  - An operation $e_0 \sqcap e_1$, the "meet" (greatest lower bound) of e_0 and e_1
    - The greatest lower bound: the greatest element e′ s.t., $e' \sqsubseteq e_0$ and $e' \sqsubseteq e_1$
  - Two named elements: $\top$ ("top") and $\bot$ ("bot"), s.t., $e \sqsubseteq \top$ and $\bot \sqsubseteq e$ for all e

- (Trivia—Complete lattice: arbitrary joins/meets, every subset rather than binary joins/meets)

Noninterference

- Static analysis handles **data** flow: ensure that no high-security data flows to a low-security output
- This is **not sufficient** to ensure noninterference, however:
- ```
  input0 = read_int()
  input1 = read_int()
  x = 0
  if (input0 < input1):
    x = 1
  output(x)
  ```

- This is called an “implicit flow,” the program’s **control flow** leaks a bit of the input!
  ```

Stating non-interference, semi-formally…

- Threat model: program inputs / outputs consist of “high security” and “low security” portions

- “Given a program **p**, it is **non-interfering** if, for any two inputs i0 / i1…
  - If i0/i1 are ***low-equivalent***, then the output p(i0) is ***low-equivalent*** to p(i1)”

Why is information flow hard?

- Requires reasoning across all possible **pairs** of program inputs!
- Looking at a single trace of the program is not enough
- It is a ***hyper***-property (property of a ***set*** of traces!), Clarkson & Schneider

```
  in(0), out(0)
  in(5), out(0)
  in(6), out(1)
  in(7), out(1)
  …

input0 = read_int()
if (input0 > 5):
  output(1)
else:
  output(0)
```

```
  in(0), out(0)
  in(5), out(0)
  in(6), out(0)
  in(7), out(0)
  …

input0 = read_int()
if (input0 > 5):
  output(0)
else:
  output(0)
```

Reasoning about Information Flow…

- Non-interference is a very strong security property
  - Variety of methods exist to reason about it: type systems, program logics, static analyses, dynamic analyses, etc.

- One basic idea: track the security of the “program counter”
  - Whenever we **branch** on high-security data, all outputs implicitly high-security
  - Joining from a branch gets us back to low security

```
input0 = read_int() // assume high security
if (input0 > 5): // program counter is high
  output(1)
else:
  output(0)
```


Even information flow just an abstraction...

```
input0 = read_int()
if (input0 > 5): // program counter is high
  sleep(100)
else:
  skip // do nothing
output(0)
```

This program is non-interfering, but time is a **side channel**

Side channels are *very tricky* to reason about

Summary Slide

- Security definitions are very tricky
  - Whether a program is “secure” depends on attack / threat modeling
    - Can you see power usage, timing, control-flow, or just outputs?

- Formal methods provide rigor, giving us one the only ways to truly make definite statements about a program’s security

- Taint tracking / program analysis useful for **data** flow
- Not enough for truly tracking **information** flow (must consider program counter)

- Thanks! Please get in touch (Kristopher Micinski, kkmicins@syr.edu)