

Automating Safe and Secure Software Development

Paul Gazzillo

Stevens Institute

pgazz.com

BUSINESS DAY

Equifax Says Cyberattack May Have Affected 143 Million in the U.S.

By TARA SIEGEL
7, 2017

Equifax, one of the three major credit bureaus, on Thursday said it had been compromised in a cyberattack that may have exposed the Social Security numbers of 143 million Americans.

The DAO Attacked: Code Issue Leads to \$60 Million Ether Theft



Michael del Castillo

Jun 17, 2016 at 14:00 UTC

Meltdown and Spectre

Vulnerabilities in modern computers leak passwords and sensitive data.

Meltdown and Spectre exploit critical vulnerabilities in modern processors. These hardware vulnerabilities allow programs to steal data which is currently processed on the computer. While programs are typically not permitted to read data from other programs, a malicious program can exploit Meltdown and Spectre to get hold of secrets stored in the memory of other running programs. This might include your passwords stored in a password manager or browser, your personal photos, emails, instant messages and even business-critical documents.

Meltdown and Spectre work on personal computers, mobile devices, and in the cloud. Depending on the cloud provider infrastructure, it might be possible to steal data from other customers.



BUSINESS DAY

Equifax Says Cyberattacks
Have Affected 142 Mill
U.S.

By TARA SIEGEL
7, 2017

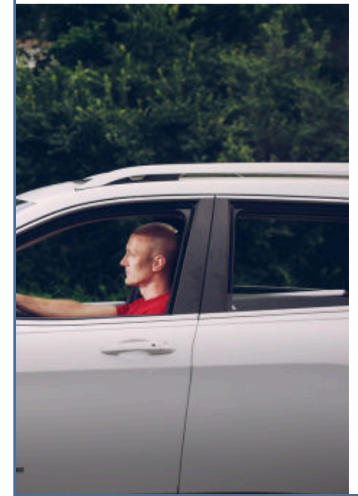
Equifax, one
Thursday the
compromise
Social Security

ANDY GREENBERG SECURITY 07.21.15 06:00 AM

HACKERS REMOTELY KILL A JEEP ON THE HIGHWAY—WITH ME IN IT

By ASHLEY WELCH / CBS NEWS / August 4, 2015, 11:29 AM

U.S. officials warn medical devices are vulnerable to hacking



ANDY GREENBERG SECURITY 07.29.15 07:00 AM

HACKERS CAN DISABLE A SNIPER RIFLE —OR CHANGE ITS TARGET



/ @ Email

medical device that could be tampered

ity issued a statement that "strongly
ue the use of Hospira's Symbiq
es are vulnerable to **cybersecurity**

to ste.
not permitted to read data from other program
ored in the memory of other running program
your personal photos, emails, instant message
the cloud. Depending on the cloud provider



y

The New York Times | <https://nyti.ms/2xSCMSr>

BUSINESS DAY

Equifax Says Cyberattacks
Have Affected 142 Mill
U.S.

ANDY GREENBERG SECURITY 07.21.15 06:00 AM

HACKERS REMOTELY KILL A JEEP ON THE HIGHWAY—WITH ME IN IT

By ASHLEY WELCH / CBS NEWS / August 4, 2015, 11:29 AM

U.S. officials warn medical

“As everything turns into a computer,
computer security becomes everything security”
- Bruce Schneier

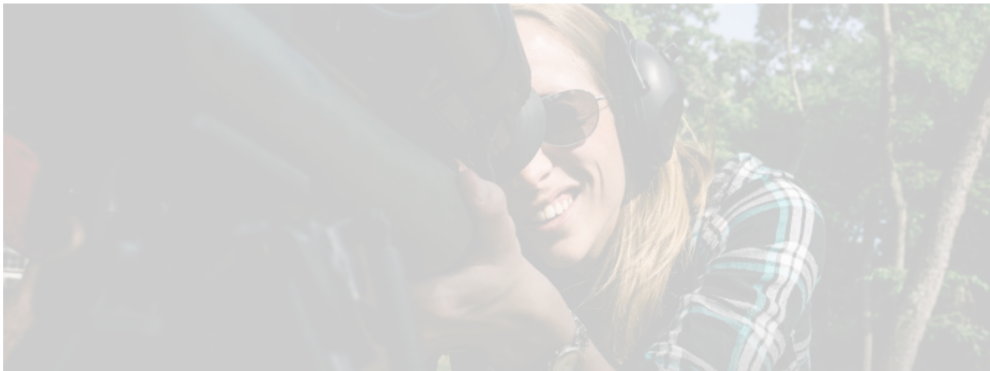
ANDY GREENBERG

HACKERS —OR CHANGE ITS TARGET

medical device that could be tampered

ity issued a statement that "strongly
the use of Hospira's Symbiq
es are vulnerable to cybersecurity

to ste
not permitted to read data from other program
ored in the memory of other running program
your personal photos, emails, instant message
the cloud. Depending on the cloud provider





Antivirus

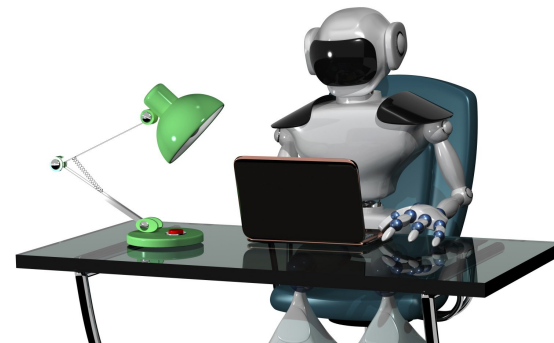


Firewalls

What Can We Do About It?

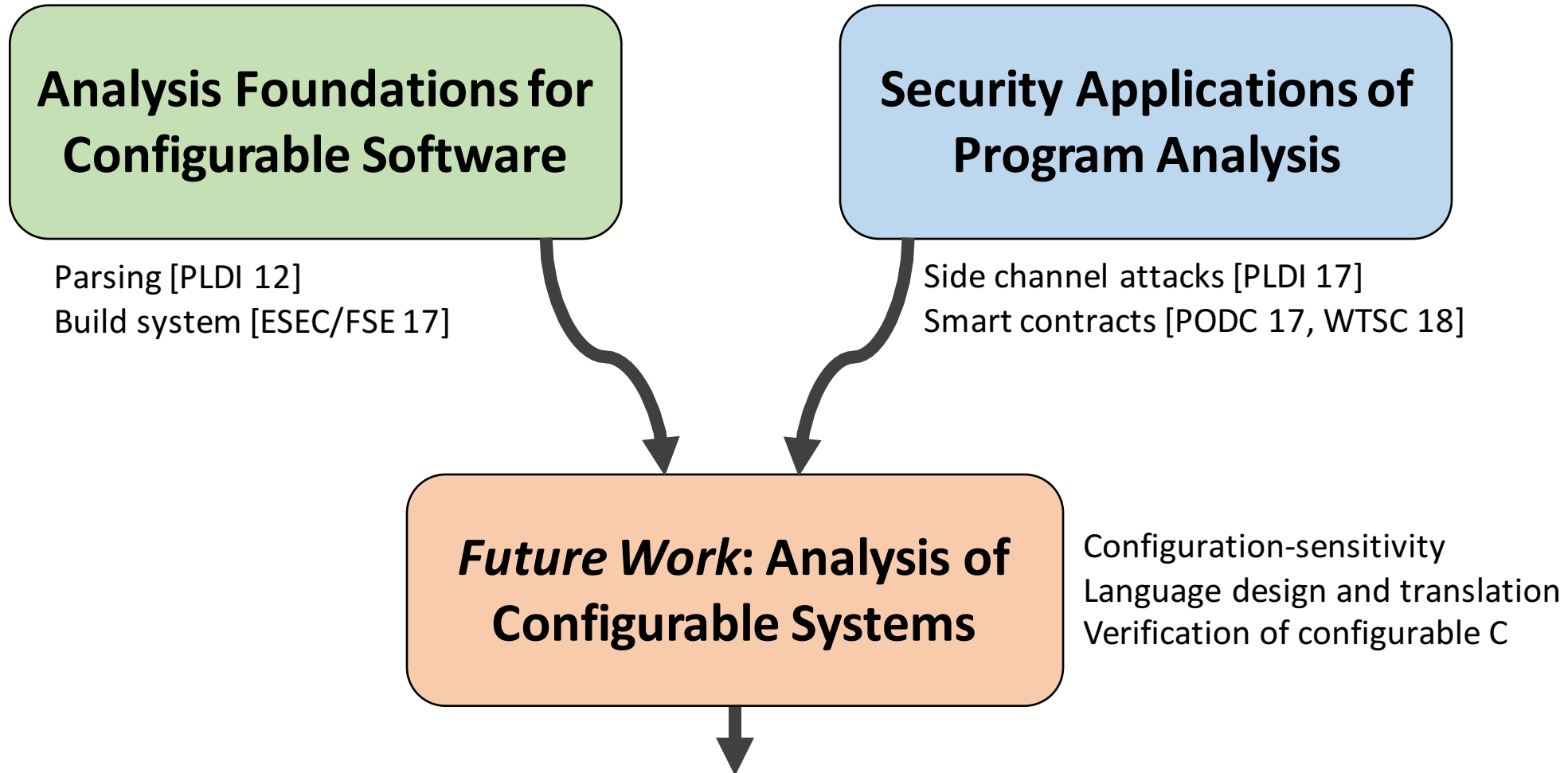


Manual code review

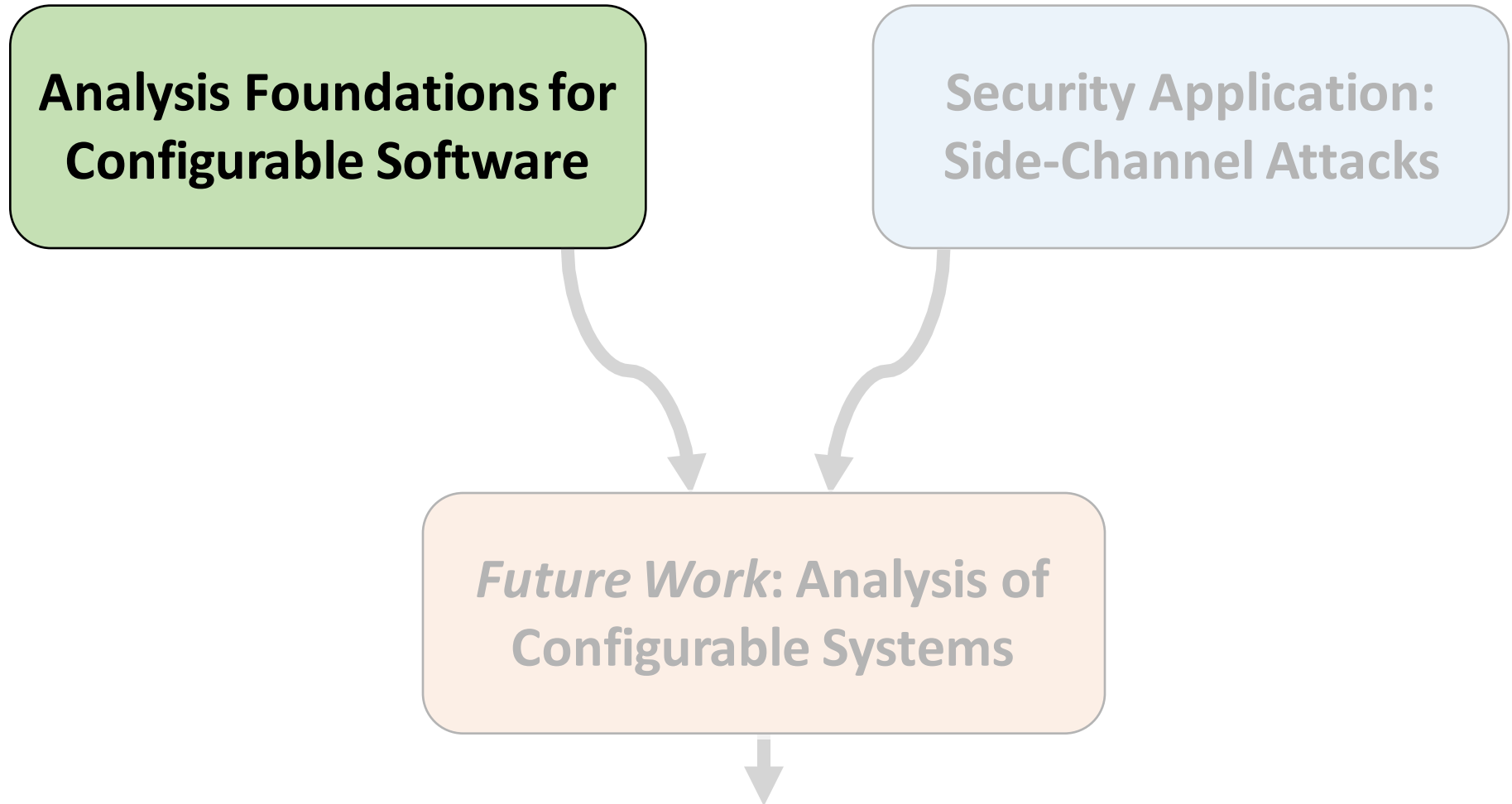


Program analysis

Overview



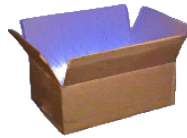
Overview



Critical Software is *Highly-Configurable*



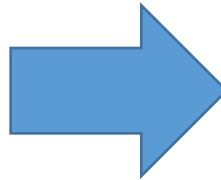
OS Kernels



IoT Toolkits



Internet Services



What Do I Mean by “Configurable”?

- Build-time configuration
 - C preprocessor and Makefiles
 - “Software product line”
- Focus on C systems
 - Linux, BusyBox, Apache, Firefox
 - 2nd place in IEEE Spectrum rankings
 - Rise of Internet-of-things

Configurable C

```
#ifdef CONFIG_OF_IRQ_DOMAIN  
void irq_add(int *ops) {  
    int irq = *ops;  
}
```

```
#endif
```

*Configuration options
tested at build-time*

```
int *ops = NULL;
```

```
#ifdef CONFIG_OF_IRQ
```

```
ops = &irq_ops;
```

```
#endif
```

```
irq_add(ops);
```


Plain C



No problems

3. This function dereferences "ops"

```
void irq_add(int *ops) {  
    int irq = *ops;  
}
```

1. Initialize "ops" pointer

```
int *ops = NULL;
```

2. Set "ops" to existing structure

```
ops = &irq_ops;
```

```
irq_add(ops);
```



Configurable C



Some configurations are fine



Some configurations have bugs

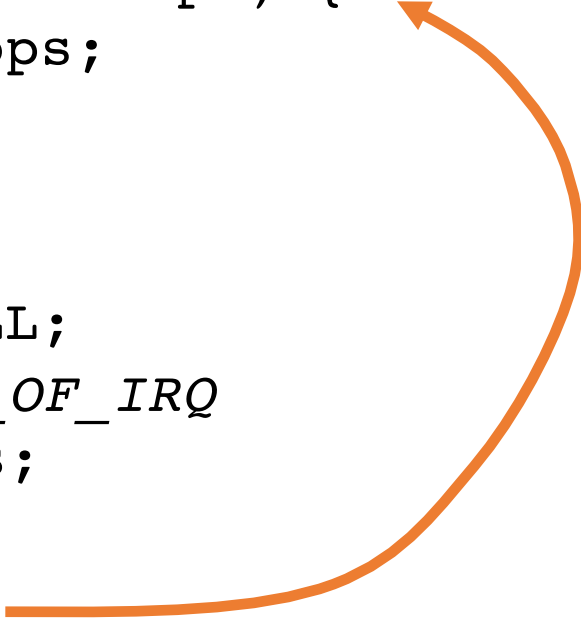
3. *Null pointer error in some configurations*

1. Initialize “ops” pointer

2. Only set in *some* configurations

```
#ifdef CONFIG_OF_IRQ_DOMAIN
void irq_add(int *ops) {
    int irq = *ops;
}
#endif

int *ops = NULL;
#ifdef CONFIG_OF_IRQ
ops = &irq_ops;
#endif
irq_add(ops);
```



Configurable C



Some configurations are fine



Some configurations have bugs

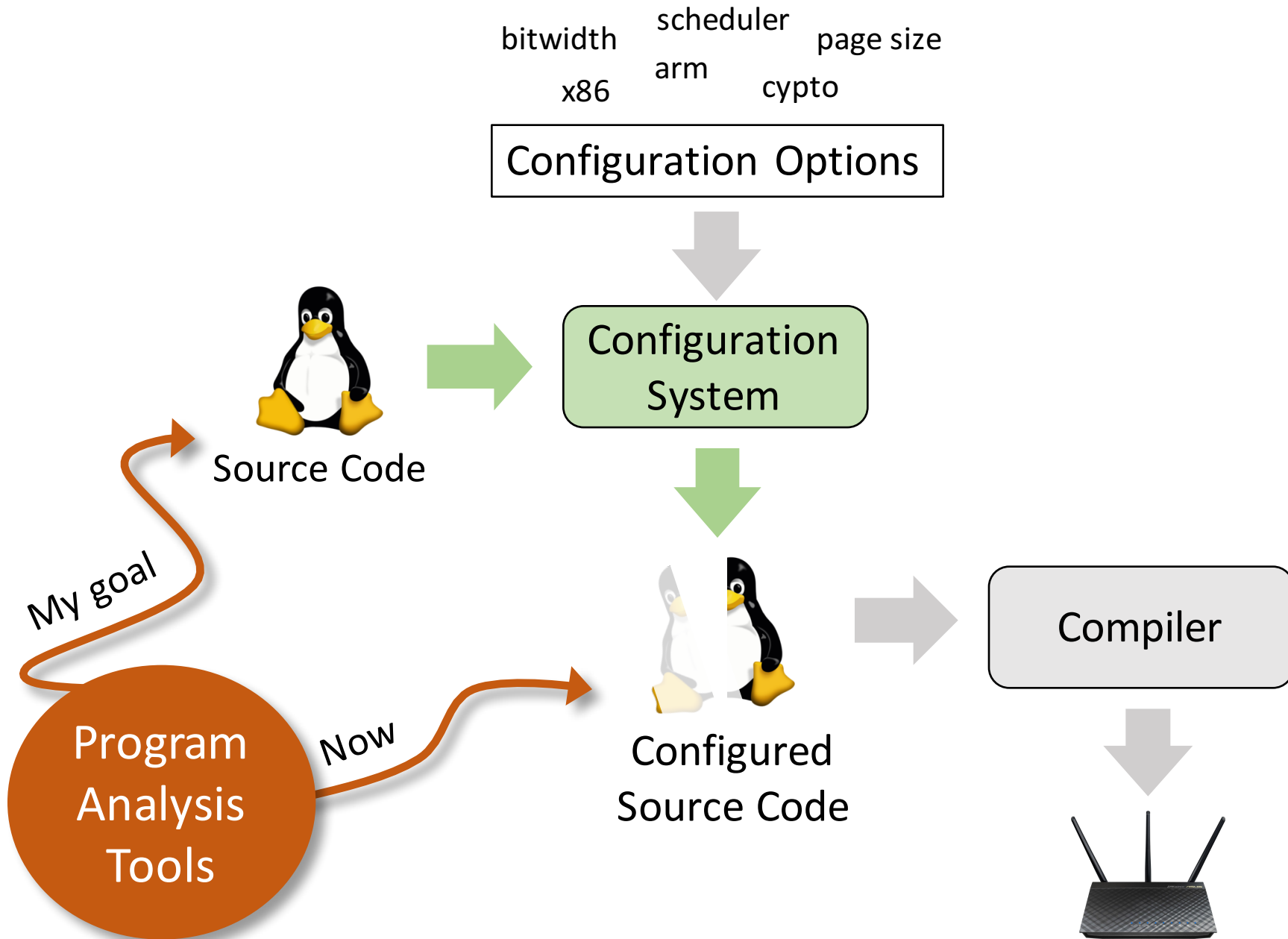
```
#ifdef CONFIG_OF_IRQ_DOMAIN
void irq_add(int *ops) {
    int irq = *ops;
}
#endif
```

```
int *ops = NULL;
#ifdef CONFIG_OF_IRQ
ops = &irq_ops;
#endif
irq_add(ops);
```

Bonus defect: undefined function breaks the build

Configurable Code is Dangerous

- Defects appear in arbitrary configurations [Abal et al ASE 14]
- Configurable code more buggy [Ferreira et al SPLC 16]
- Debugging is harder [Melo et al ICPC 17]



Check Configurations One-at-a-Time?

System	SLoC	Options	Configurations
axTLS web server	3k	94	2 trillion
BusyBox embedded toolkit	17k	993	2^{827}
Linux kernel	12mil	14,000+	$< 2^{14,000}$

Estimated # atoms in the universe: 2^{266}

Check Configurations One-at-a-Time?

System	SLoC	Options	Configurations
axTLS web server	2k	04	2 trillion
BusyBox embedded			
Linux kernel	12mil	14,000+	$< 2^{14,000}$

Goal
Analyze All Configurations
Simultaneously

Estimated # atoms in the universe: 2^{266}

Analysis Front-End

Which source files comprise the program

Build System

Parser

Produces a tree representation of the source

Bug finders

Security analysis

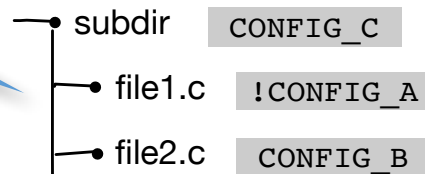
Code browsers

Configuration-Aware Front-End

Finds source files for each configuration

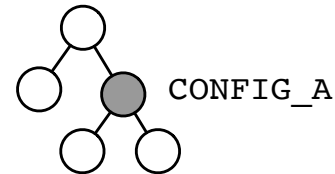
Build System

Kmax [ESEC/FSE 17]



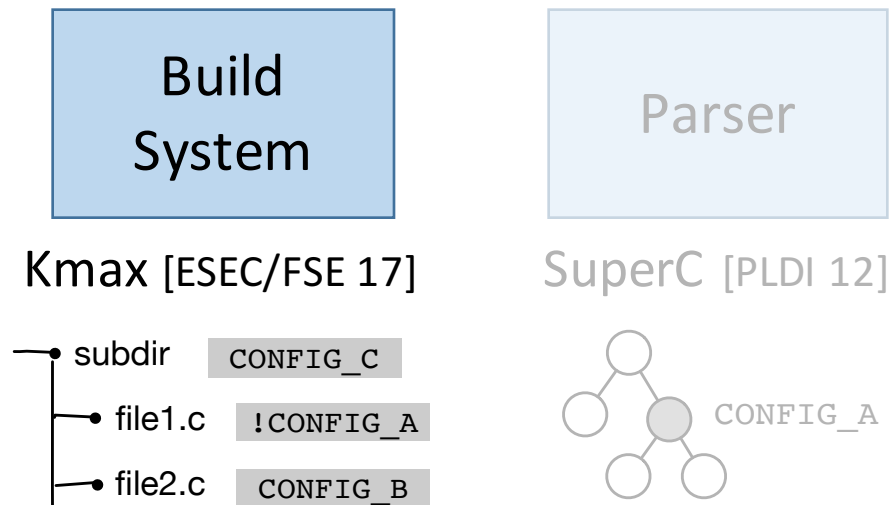
Parser

SuperC [PLDI 12]



Produces an AST for all configurations

Build System Analysis with Kmax



Why Is Finding All Source Files Hard?

Conditionals define configurations

Adds source files to obj-y

```
obj-y := fork.o  
ifeq ($(CONFIG_A),y)
```

```
    BITS := 32
```

```
else
```

```
    BITS := 64
```

```
endif
```

```
obj-$(CONFIG_B) += probe_$(BITS).o
```

String concatenation

Generated *variable* names!

Brute Force: Run the Makefile for Each Configuration

```
obj-y := fork.o
ifeq ($(CONFIG_A),y)
    BITS := 32
else
    BITS := 64
endif
obj-$(CONFIG_B) += probe_$(BITS).o
```

Exponential # of configurations

Duplicate information

Configuration Options		Final State
CONFIG_A	CONFIG_B	obj-y
on	on	fork.o probe_32.o
on	off	fork.o
off	on	fork.o probe_64.o
off	off	fork.o

Kmax Finds All Configurations Efficiently

- Symbolic evaluation of conditional expressions
 - Compact representation of configurations (BDDs)

`CONFIG_B \wedge \neg CONFIG_A`

- Concrete evaluation of strings

- Exact file names
- Deduplication with symbolic comparisons

`["probe_32.o" if BITS==32 \wedge CONFIG_B ,
 "probe_64.o" if BITS==64 \wedge CONFIG_B]`

```

obj-y := fork.o
ifeq ($(CONFIG_A),y)
    BITS := 32
else
    BITS := 64
endif
obj-$(CONFIG_B) += probe_$(BITS).o

```

Kmax

Better than
exponential

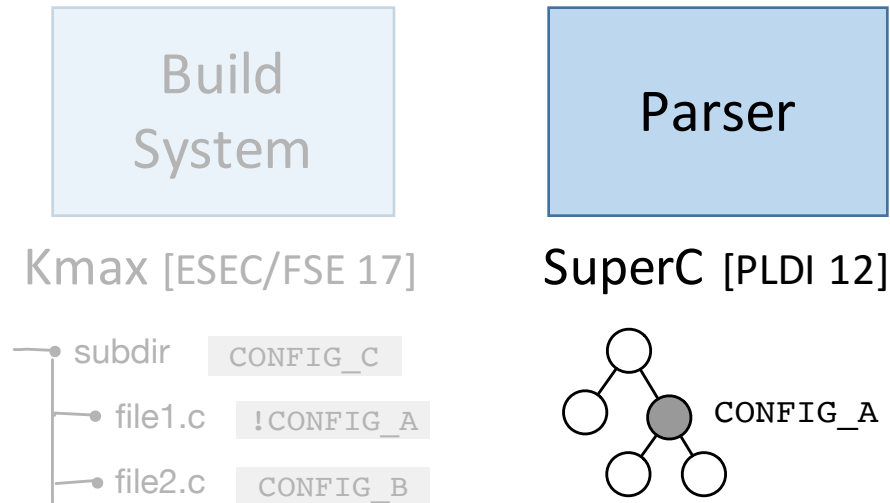
No more
duplication

obj-y's values	
Configurations	Source File
All	fork.o
CONFIG_A \wedge CONFIG_B	probe_32.o
\neg CONFIG_A \wedge CONFIG_B	probe_64.o

Summary of Kmax

- Finds source files for all configurations efficiently
- What can we do with this information?
 - Find set of files for bug finders
 - Eliminate dead code
 - Find the configurations needed to test certain files
 - Determine a patch's impact on the whole system
 - (Google summer of code project)

Parsing All of C with SuperC



Isn't Parsing a Solved Problem?

- C programs written in *two* languages: preprocessor and C itself
- Macros expand to arbitrary C fragments

```
#define for_each_class(c) \
    for (c = highest_class; c; c = c->next)
```

- Directives appear between arbitrary C fragments

```
#ifdef CONFIG_INPUT_MOUSEDEV_PSAUX
    if (imajor(inode) == 10)
        i = 31;
    else
#endif
        i = iminor(inode) - 32;
```

SuperC to the Rescue!

- Configuration-preserving preprocessor
 - Expands macros and includes headers
 - But preserves conditionals
- Fork-merge parser
 - Invokes multiple subparsers at ifdefs
 - Combine subparsers after ifdefs

Cond

```
#ifdef CONFIG_64BIT
#  define BITS_PER_LONG 64
#else
#  define BITS_PER_LONG 32
#endif
```

processor!

Function-like macros

Macro definitions

Static co

Conditional expressions

Stringification

Token-pasting

__le ## 32

→ __le32

Includes

Conditional

Macro expands to
conditional

One operator:
Two operations

```
__le ## BITS_PER_LONG
```

Hoist conditional
around token-paste

```
__le ##  
#ifdef CONFIG_64BIT  
    64  
#else  
    32  
#endif
```

```
#ifdef CONFIG_64BIT  
    __le ## 64  
#else  
    __le ## 32  
#endif
```

SuperC to the Rescue!

- Configuration-preserving preprocessor
 - Expands macros and includes headers
 - But preserves conditionals!
- Fork-merge parser
 - Manages multiple *subparsers*
 - Each subparser handles a different configuration

Parsing in A

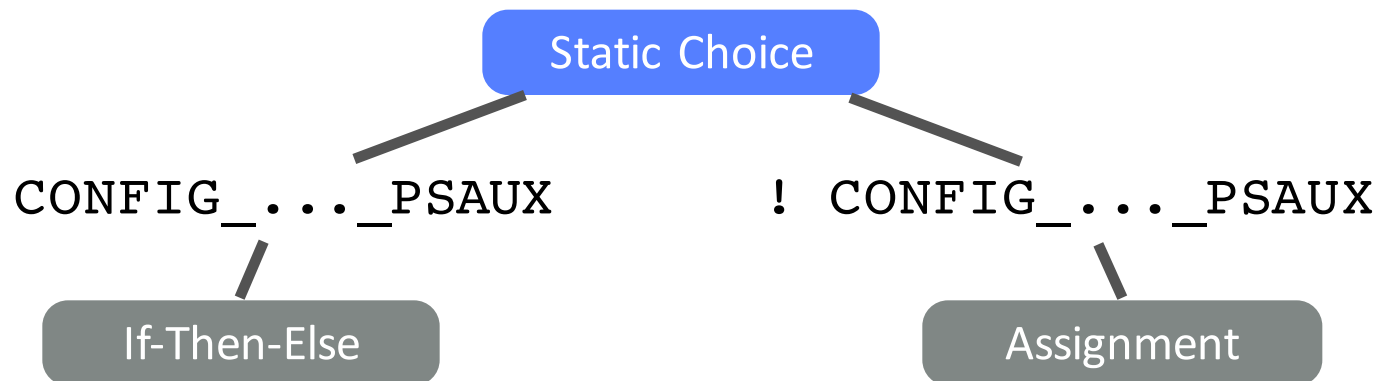
(1) Fork subparsers on conditional

(2) Parse the *entire* if-then-else

(3) Parse *just* the assignment

(4) Merge and create the static choice node

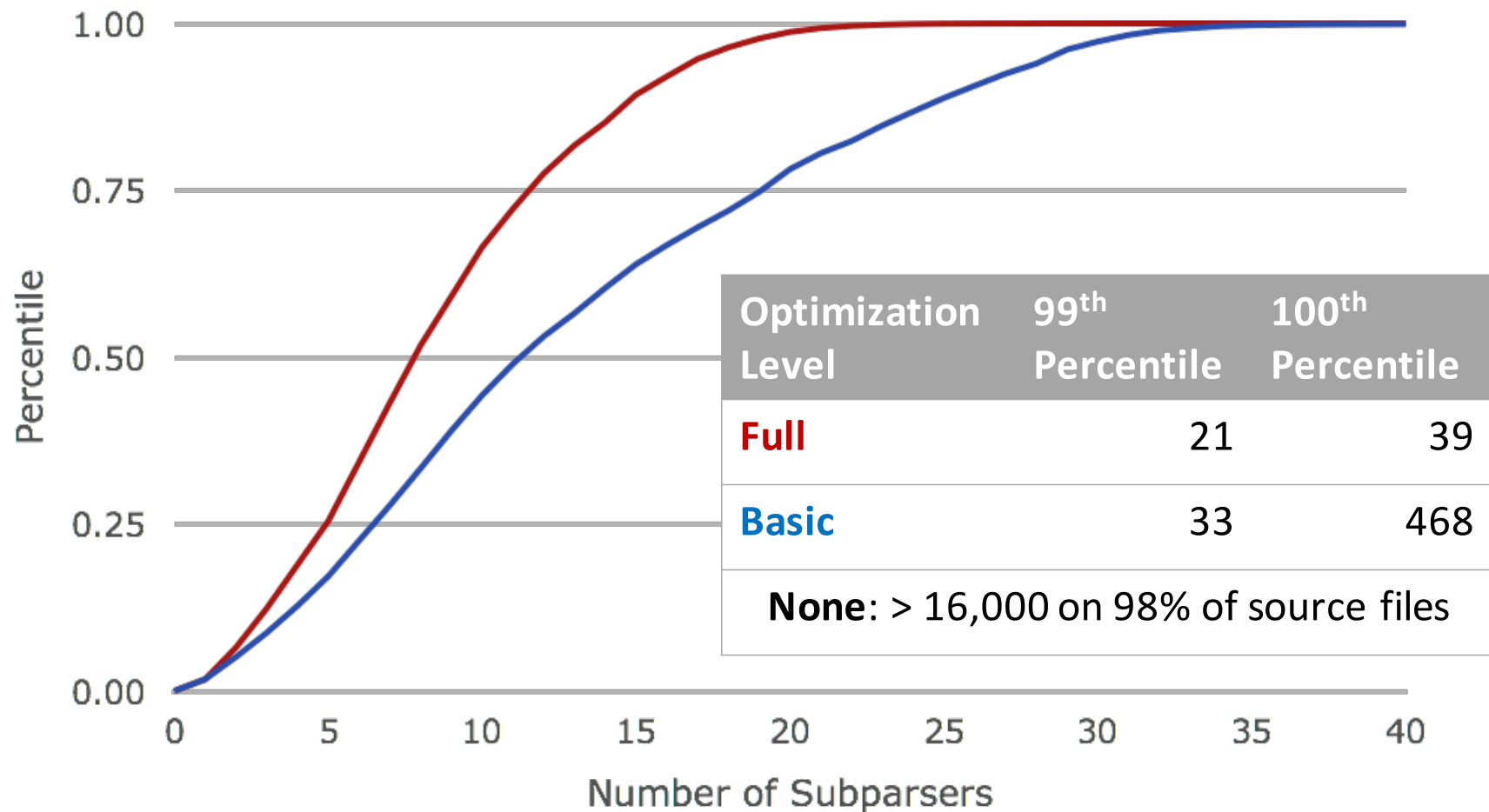
```
#ifdef CONFIG_INPUT_MOUSEDEV_PSAUX
    if (imajor(inode) == 10)
        i = iminor(inode) - 32,
        if (i >= ...
```



SuperC Performance

- How many simultaneous configurations?
- Tested on entire Linux x86 kernel source
 - Thousands of preprocessor conditionals deeply nested
 - No more than 10s per source file
- Novel algorithmic optimizations
 - Dramatically reduce configuration explosion
 - Without resorting to incomplete heuristics

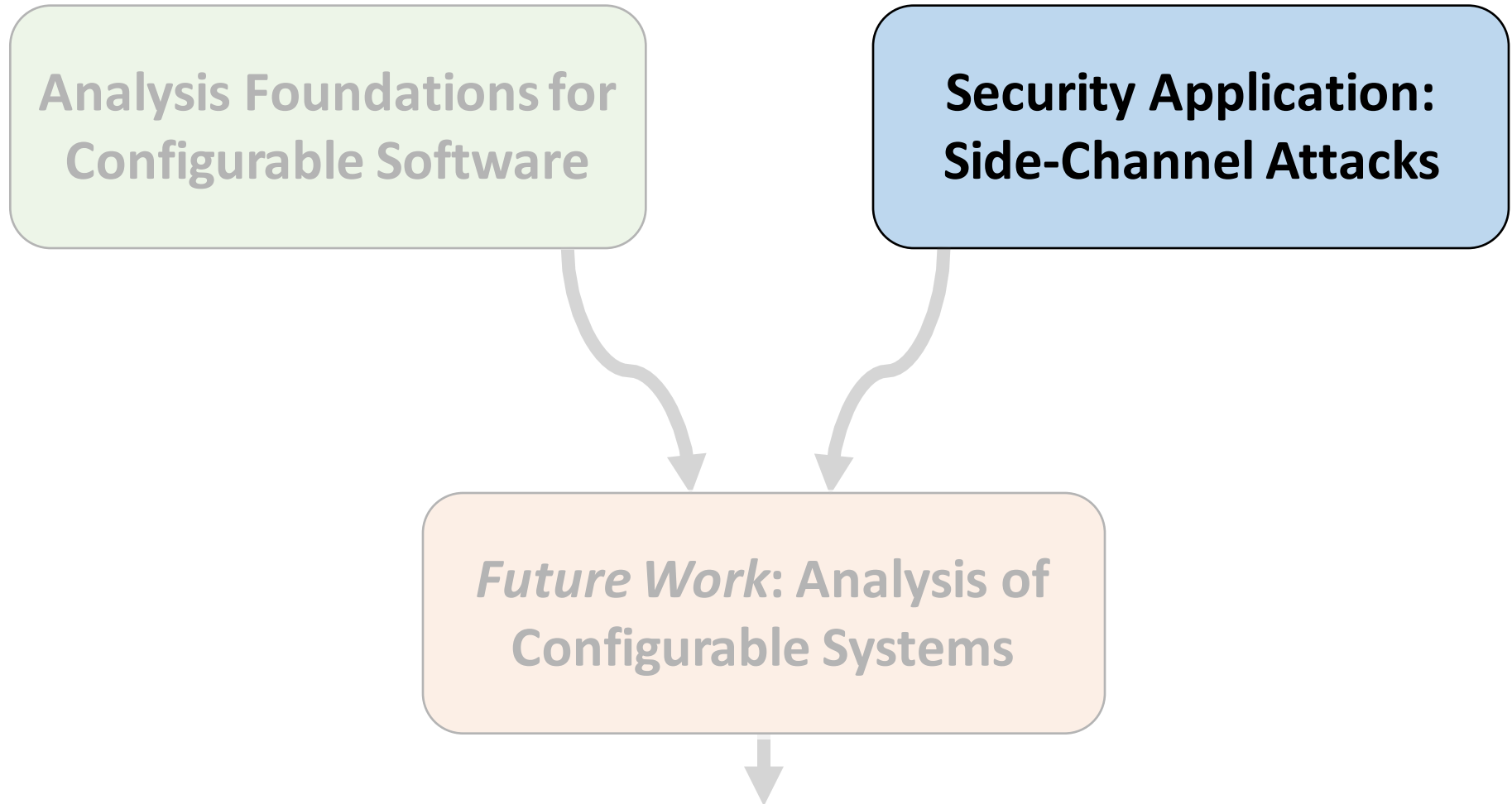
How Many Simultaneous Configurations?



Conclusion

- Program analysis tools work on one configuration
- Exhaustive search infeasible
- Analyze all configurations simultaneously
 - SuperC parses all configurations
 - Kmax finds set of source files for all configurations
- Helps software tools scale to large configurable systems

Overview



Meltdown and Spectre

Vulnerabilities in modern computers leak passwords and sensitive data.

Meltdown and Spectre exploit critical vulnerabilities in modern processors. These hardware vulnerabilities allow programs to steal data which is currently protected. If a malicious program can exploit these vulnerabilities, it might include your passwords and even business-critical data.

Meltdown and Spectre work on all modern processors. If your infrastructure, it might be possible to exploit these vulnerabilities.

IoT Goes Nuclear: Creating a ZigBee Chain Reaction

Eyal Ronen, Colin O'Flynn, Adi Shamir and Achi-Or Weingarten

Creating an IoT Chain Reaction

Within the next few years, IoT devices will densely populate our environment. In this paper we describe how to create an adjacent IoT device chain reaction that will spread exponentially. We show how a nuclear chain reaction of compatible IoT devices can be created. In particular, we develop a platform.

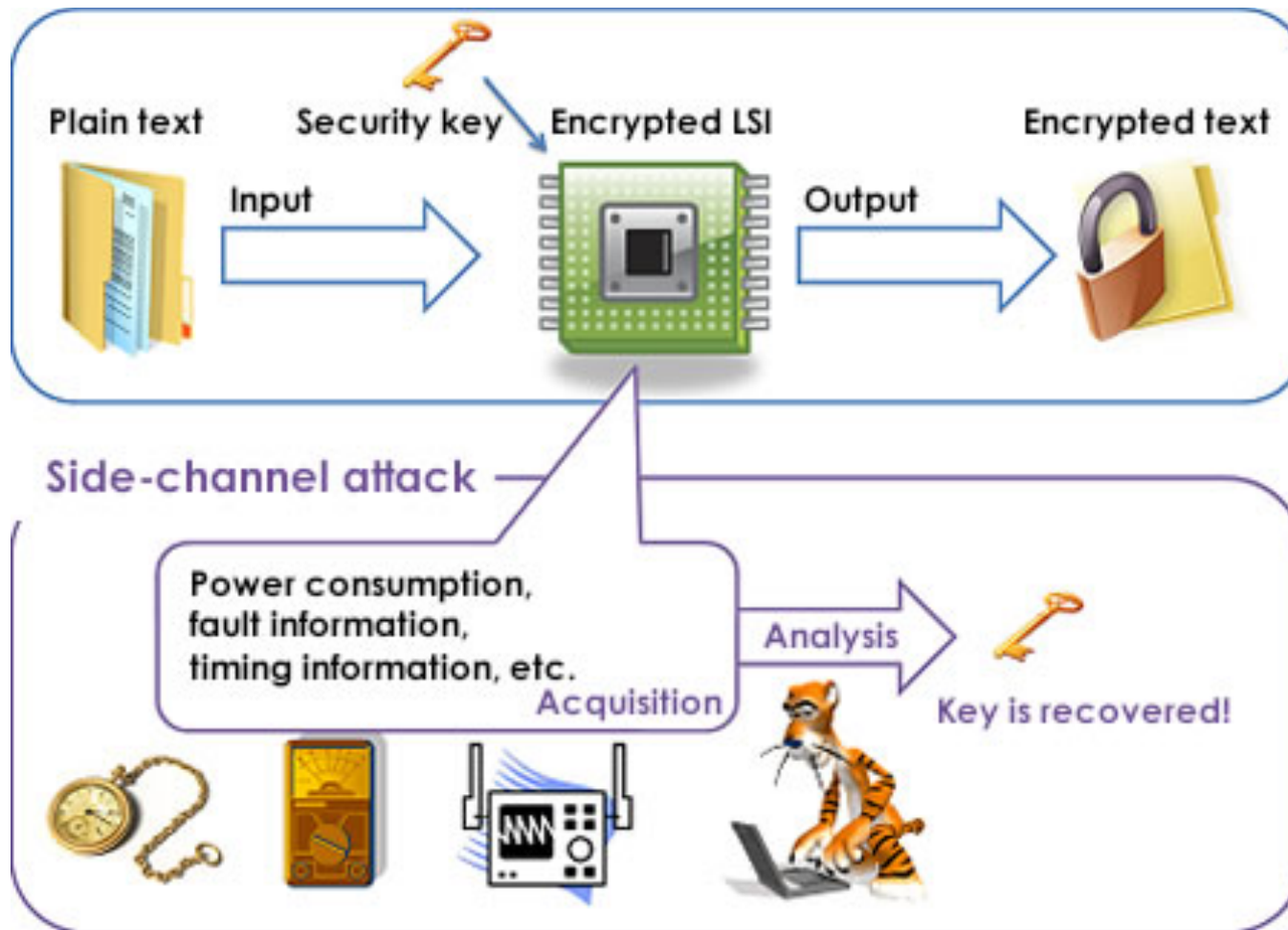
Differential Power Analysis

Paul Kocher, Joshua Jaffe, and Benjamin Jun

Cryptography Research, Inc.
607 Market Street, 5th Floor
San Francisco, CA 94105, USA.
<http://www.cryptography.com>

E-mail: {paul,josh,ben}@cryptography.com.

Abstract. Cryptosystem designers frequently assume that secrets will be manipulated in closed, reliable computing environments. Unfortunately, actual computers and microchips leak information about the operations they process. This paper examines specific methods for analyzing power consumption measurements to find secret keys from tamper



Source: <http://www.togawa.cs.waseda.ac.jp/English/research/system/system.html#sca>

→ Changing password for paul.
→ (current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:

```
boolean chpass(real_pwd, input_pwd, new_pwd, new_pwd_confirm) {  
    int correct_chars = 0;  
    for(int i = 0; i < input_pwd.length; i++) {  
        if(i < real_pwd.length && real_pwd[i] == input_pwd[i])  
            correct_chars += 1;  
        else  
            return false;  
    }  
  
    boolean matches = true;  
    if(new_pwd.length == new_pwd_confirm.length) {  
        for (int i = 0; i < new_pwd.length; i++)  
            matches = matches && (new_pwd[i] == new_pwd_confirm[i]);  
    } else  
        matches = false;  
  
    return (correct_chars == real_pwd.length) && matches;  
}
```

Number of loop iterations reveals correct character count

Side Channels Reduce Search Space

Password: **hello**

Brute force

(26 character options)⁵ characters
= 11,881,376 guesses

With timing channel

Char	Iters	Guesses
h	1	26
e	2	26
l	3	26
l	4	26
o	5	26

26 character options x 5 characters
= 130 guesses

```
Changing password for paul.  
(current) UNIX password:  
Enter new UNIX password:  
Retype new UNIX password:
```

```
boolean chpass(real_pwd, input_pwd, new_pwd, new_pwd_confirm) {  
    int correct_chars = 0;  
    for(int i = 0; i < input_pwd.length; i++) {  
        if(i < real_pwd.length && real_pwd[i] == input_pwd[i])  
            correct_chars += 1;  
        else  
            return false; correct_chars += 0;  
    }  
  
    boolean matches = true;  
    if(new_pwd.length == new_pwd_confirm.length) {  
        for (int i = 0; i < new_pwd.length; i++)  
            matches = matches && (new_pwd[i] == new_pwd_confirm[i]);  
    } else  
        matches = false;  
  
    return (correct_chars == real_pwd.length) && matches;  
}
```

Loop iterations now
independent of
correct characters

Timing Channel Freedom

$$\begin{aligned} &\forall \pi_1, \pi_2. \\ &\text{in}(\pi_1)[\text{low}] = \text{in}(\pi_2)[\text{low}] \\ &\quad \Rightarrow \\ &\text{time}(\pi_1) = \text{time}(\pi_2) \pm c \end{aligned}$$

This means low input values for each trace

- Running time does not depend on secret
- Any two traces have roughly same running time for same low input
- A 2-safety property, i.e., must relate pairs of traces to prove property

Reframe Timing Channel Freedom

$$\begin{aligned} & \forall \pi_1, \pi_2. \\ & \text{in}(\pi_1)[\text{low}] = \text{in}(\pi_2)[\text{low}] \\ & \Rightarrow \\ & \text{time}(\pi_1) = \text{time}(\pi_2) \pm c \end{aligned}$$



$$\begin{aligned} & \exists t. \forall \pi. \\ & \text{time}(\pi) = t(\text{in}(\pi)[\text{low}]) \pm c \end{aligned}$$

- Function of public inputs *only*
- Non-relational: in terms of one trace
- Implies timing channel freedom, a relational property

Prove with Running Time Analysis

```
for (int i = 0; i < new_pwd.length; i++) {  
    matches = matches && (new_pwd[i] == new_pwd_confirm[i]);  
}
```

Static Running
Time Analysis

[Gulwani et al, PLDI 09;
CAV 09; POPL 09]

$\text{time}(\pi) = t(\text{in}(\pi)[\text{new_pwd}]) = \text{new_pwd.length}$

- Finds running time function t
- Implies timing channel freedom

Finding t Is Hard

- Programs can have nested conditionals and loops
 - Many branches on public inputs
 - At any program point
 - In loop headers
- t can be *piecewise* with complex cases

$$t = \left\{ \dots \left\{ \dots \left\{ \dots \right. \right. \right.$$

$$t = \begin{cases} \text{input_pwd.len} * 2 + \text{new_pwd.len} * 2 + 3 & \text{if } \text{new_pwd.len} == \text{new_pwd_confirm.len} \\ \text{input_pwd.len} * 2 + 4 & \text{if } \text{new_pwd.len} != \text{new_pwd_confirm.len} \end{cases}$$

```
boolean chpass(real_pwd, input_pwd, new_pwd, new_pwd_confirm)
{
    int correct_chars = 0;
    for(int i = 0; i < input_pwd.length; i++) {
        if(i < real_pwd.length && real_pwd[i] == input_pwd[i])
            correct_chars += 1;
        else
            correct_chars += 0;
    }

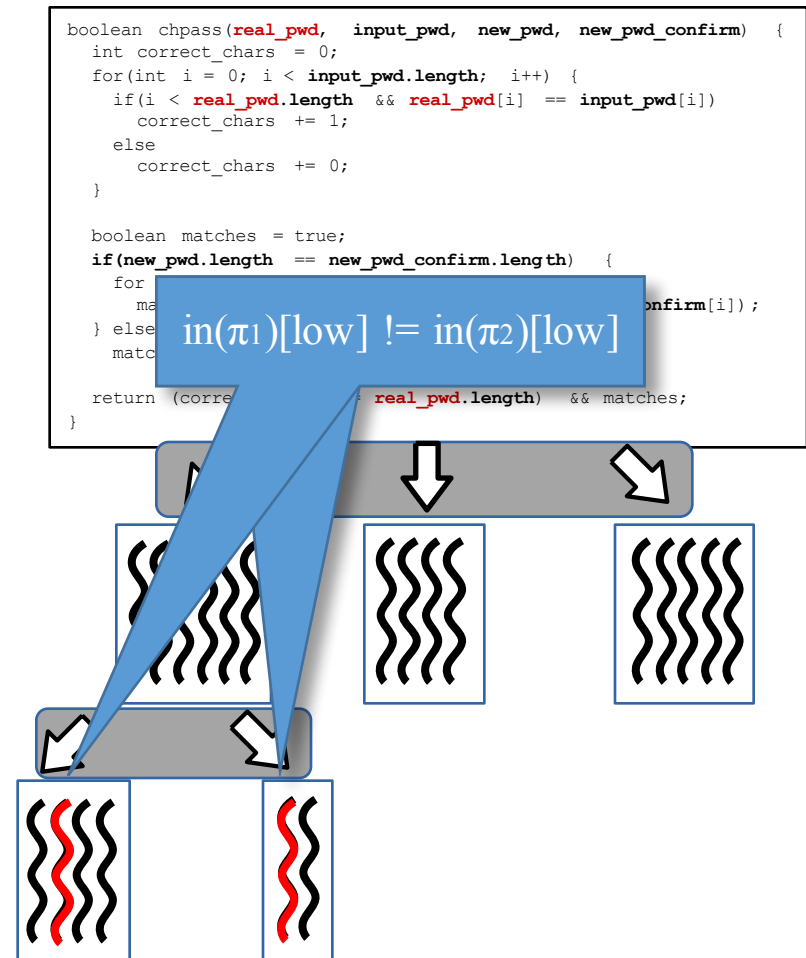
    boolean matches = true;
    if(new_pwd.length == new_pwd_confirm.length) {
        for (int i = 0; i < new_pwd.length; i++)
            matches = matches && (new_pwd[i] == new_pwd_confirm[i]);
    } else
        matches = false;

    return (correct_chars == real_pwd.length) && matches;
}
```

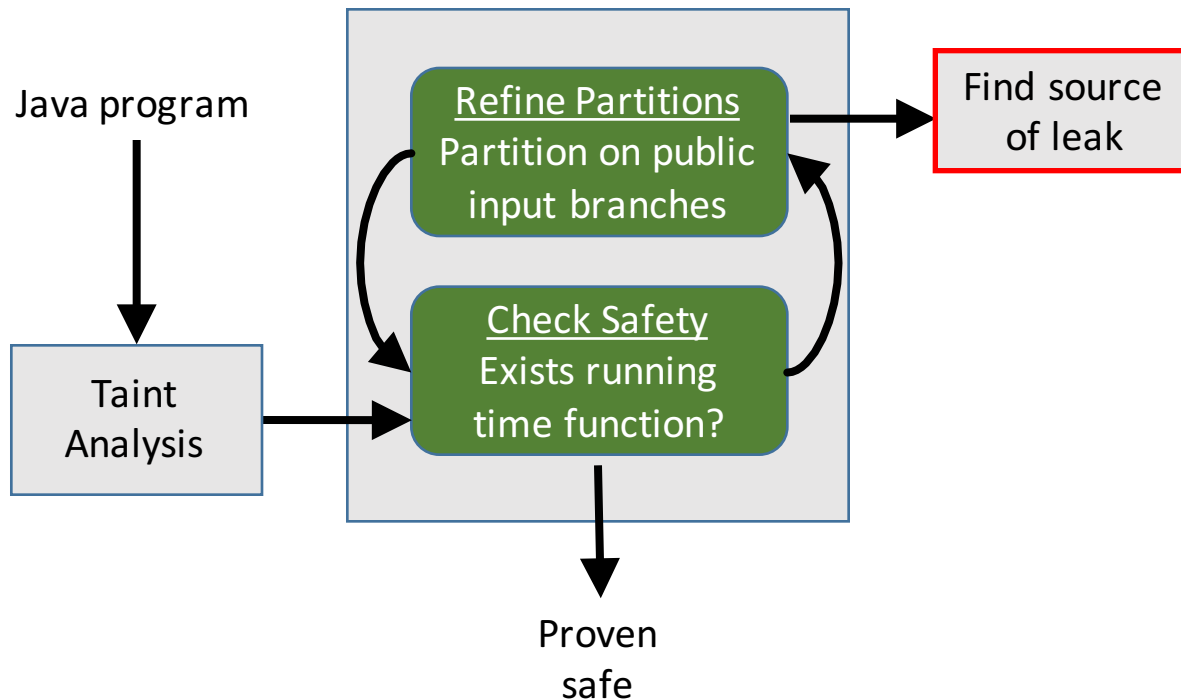
- Piecewise function with complex cases
- Running time analysis can't do this well

Partition the Program

- Prove freedom of partitions alone
- Must choose partitions carefully
- Prove safety of partitions *separately*
- Implies safety of complete program

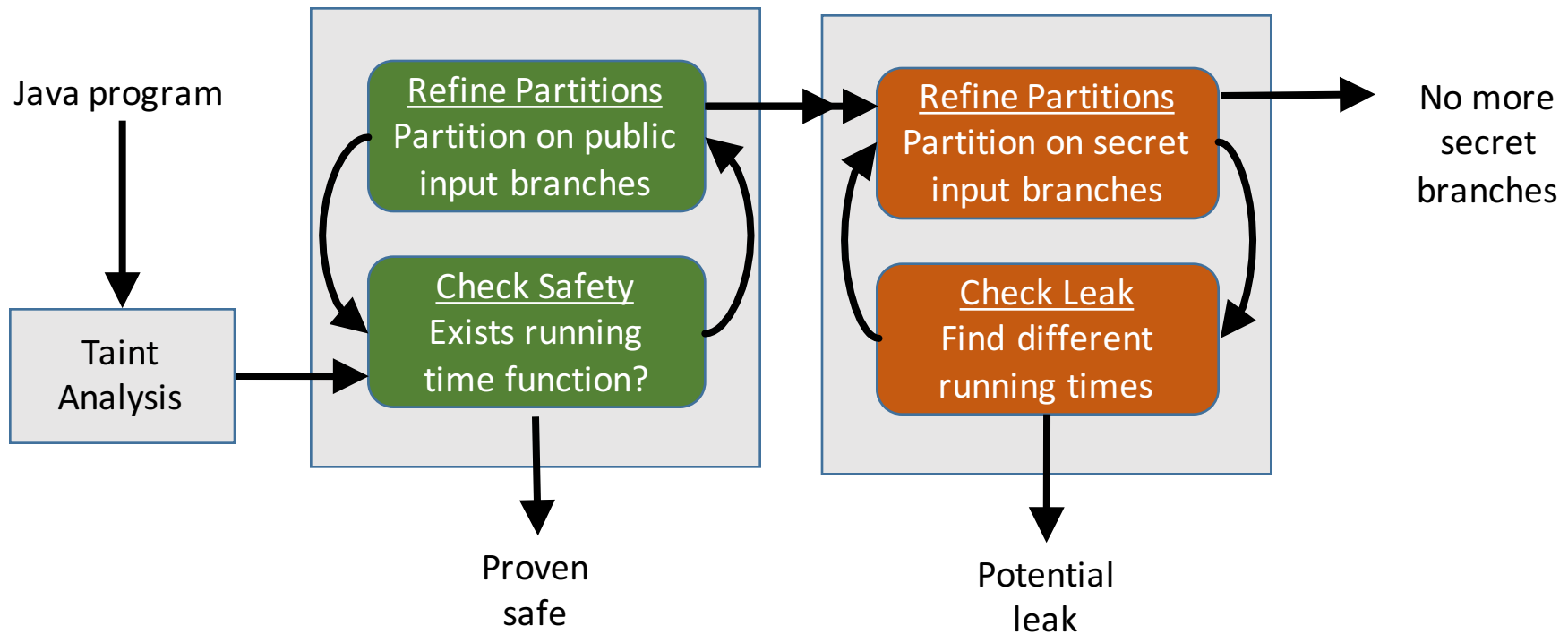


Safety Proving Algorithm



- Use taint analysis to get non-secret branches
- Use static running time bounds analysis
- Iteratively partition and check safety
- Continue partitioning until all partitions safe
- Find source of leak

Timing Channel Detection



Evaluation

- Algorithm implemented in the Blazer tool
 - Static running time analysis built on an abstract interpreter
- Usage scenario
 - Lightweight pre-analysis to narrow down suspicious methods
 - Apply heavyweight blazer to suspicious methods
- Benchmarks: several lines to several dozen lines
 - MicroBench – 12 small, simple examples
 - STAC - 6 extracted from DARPA challenge programs
 - Literature – 6 adapted from literature

Benchmark	Size	Safety Time (s)	w/Attack Time (s)
<i>MicroBench</i>			
array_safe	16	1.60	–
array_unsafe	14	0.16	0.70
loopBranch_safe	15	0.23	–
loopBranch_unsafe	15	0.65	1.54
nosecret_safe	7	0.35	–
notaint_unsafe	9	0.28	1.77
sanity_safe	10	0.63	–
sanity_unsafe	9	0.30	0.58
straightline_safe	7	0.21	–
straightline_unsafe	7	22.20	28.49
unixlogin_safe	16	0.86	–
unixlogin_unsafe	11	0.77	1.27
<i>STAC</i>			
modPow1_safe	18	1.47	–
modPow1_unsafe	58	218.54	464.52
modPow2_safe	20	1.62	–
modPow2_unsafe	106	7813.68	31758.92
pwdEqual_safe	16	2.70	–
pwdEqual_unsafe	15	1.30	2.90
<i>Literature</i>			
gpt14_safe	15	1.43	–
gpt14_unsafe	26	219.30	1554.64
k96_safe	17	0.70	–
k96_unsafe	15	1.29	3.14
login_safe	18	6.54	–
login_unsafe	17	4.40	9.10

- Size in basic blocks
- Time to prove safety
 - Average of 5 runs
- If not safe, time to prove attack
 - Average of 5 runs
- A few seconds or less for most benchmarks
 - 22.20s at most for safety proving

Proved safety or leak for all.

Benchmark	Size	Safety Time (s)	w/Attack Time (s)
<i>MicroBench</i>			
array_safe	16	1.60	–
array_unsafe	14	0.16	0.70
loopBranch_safe	15	0.23	–
loopBranch_unsafe	15	0.65	1.54
nosecret_safe	7	0.35	–
notaint_unsafe	9	0.28	1.77
sanity_safe	10	0.63	–
sanity_unsafe	9	0.30	0.58
straightline_safe	7	0.21	–
straightline_unsafe	7	22.20	28.49
unixlogin_safe	16	0.86	–
unixlogin_unsafe	11	0.77	1.27
<i>STAC</i>			
modPow1_safe	18	1.47	–
modPow1_unsafe	58	218.54	464.52
modPow2_safe	20	1.62	–
modPow2_unsafe	106	7813.68	31758.92
pwdEqual_safe	16	2.70	–
pwdEqual_unsafe	15	1.30	2.90
<i>Literature</i>			
gpt14_safe	15	1.43	–
gpt14_unsafe	26	219.30	1554.64
k96_safe	17	0.70	–
k96_unsafe	15	1.29	3.14
login_safe	18	6.54	–
login_unsafe	17	4.40	9.10

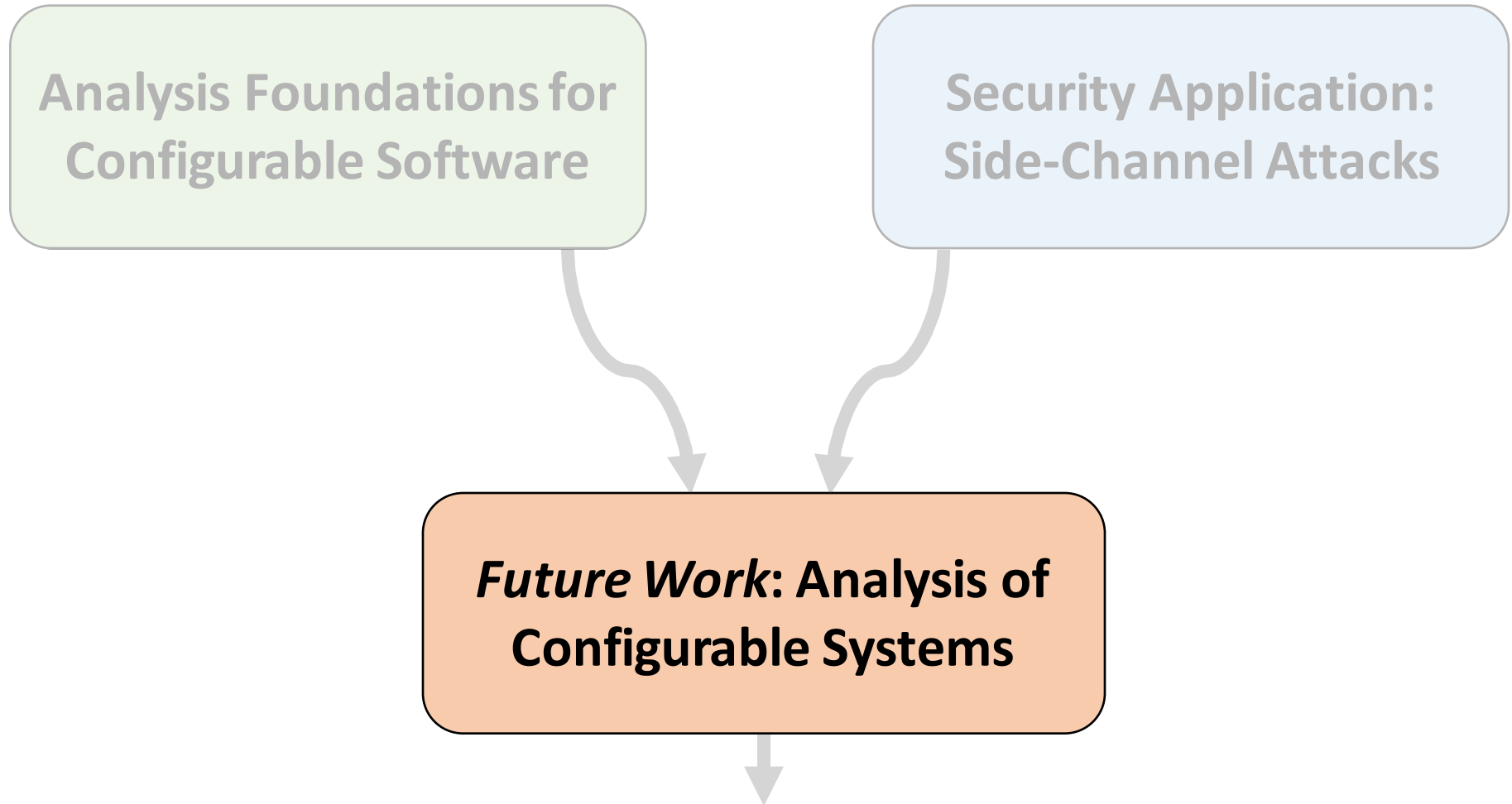
Scalability of Leak Identification

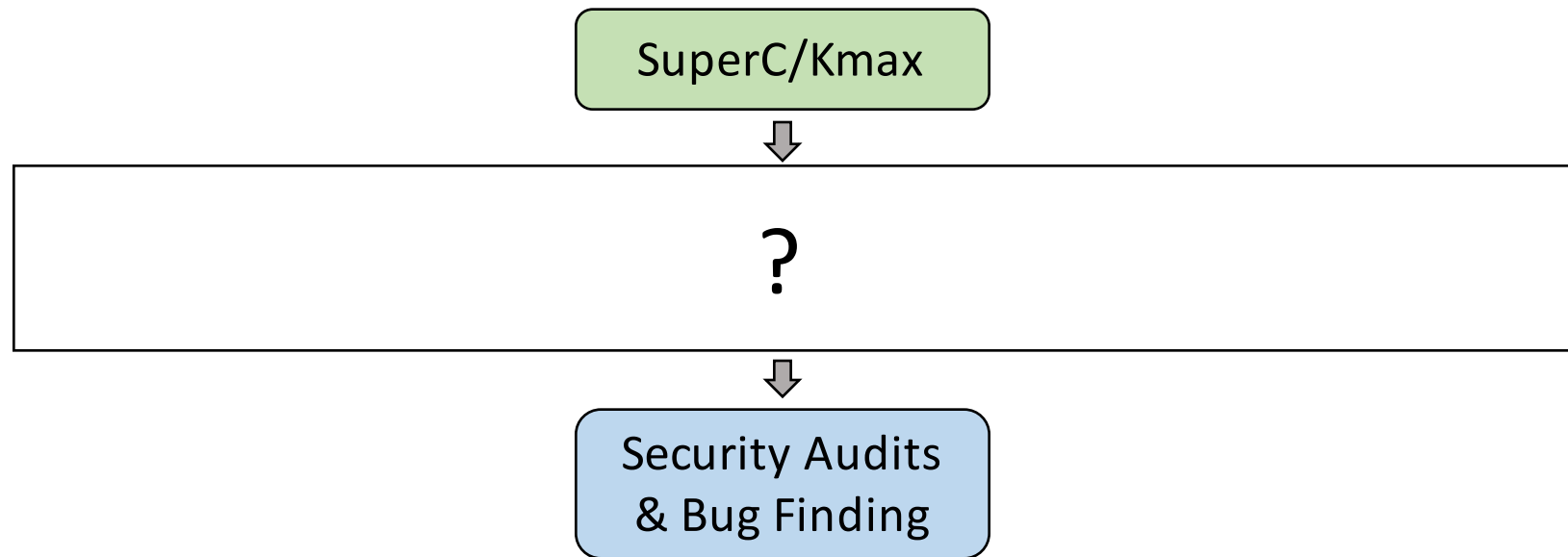
- Notable outliers
- Minutes or hours
- Related to block size
- Likely due to many partitions

Summary

- Side channel attacks leak secrets indirectly
- Prove freedom from timing leaks or identify leak
- Static analysis approach is highly precise

Overview



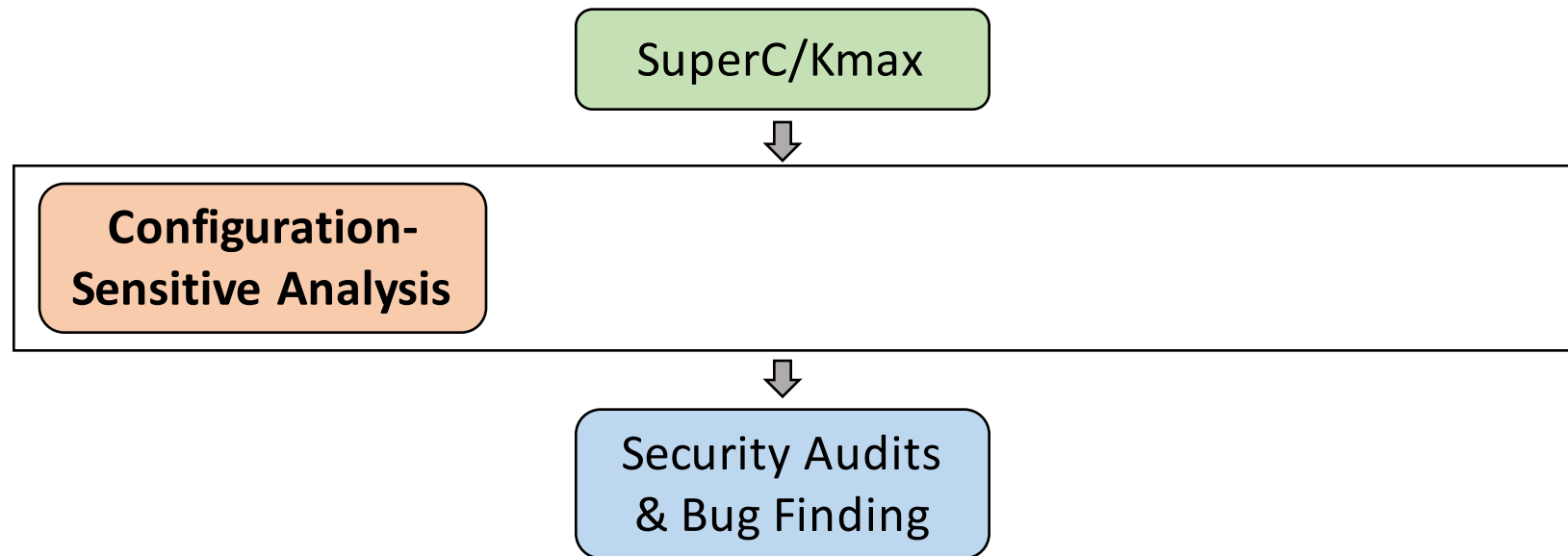


Vision

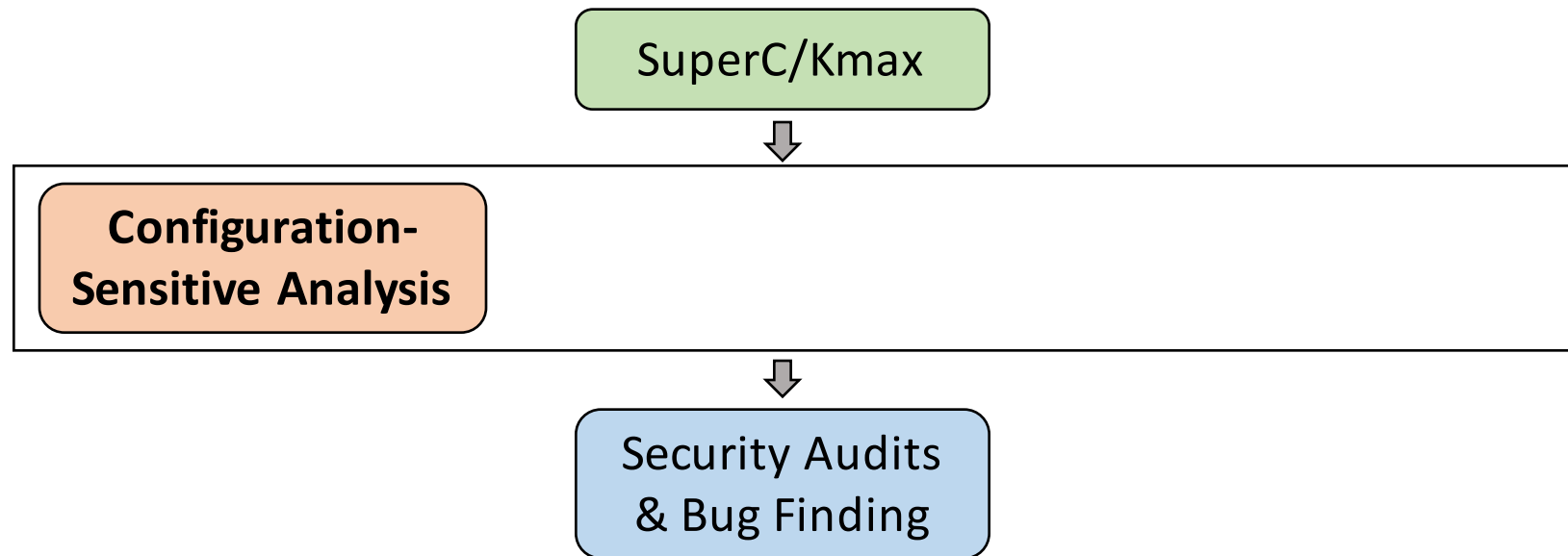
*Elevate program analysis
to configurable code.*

Impact

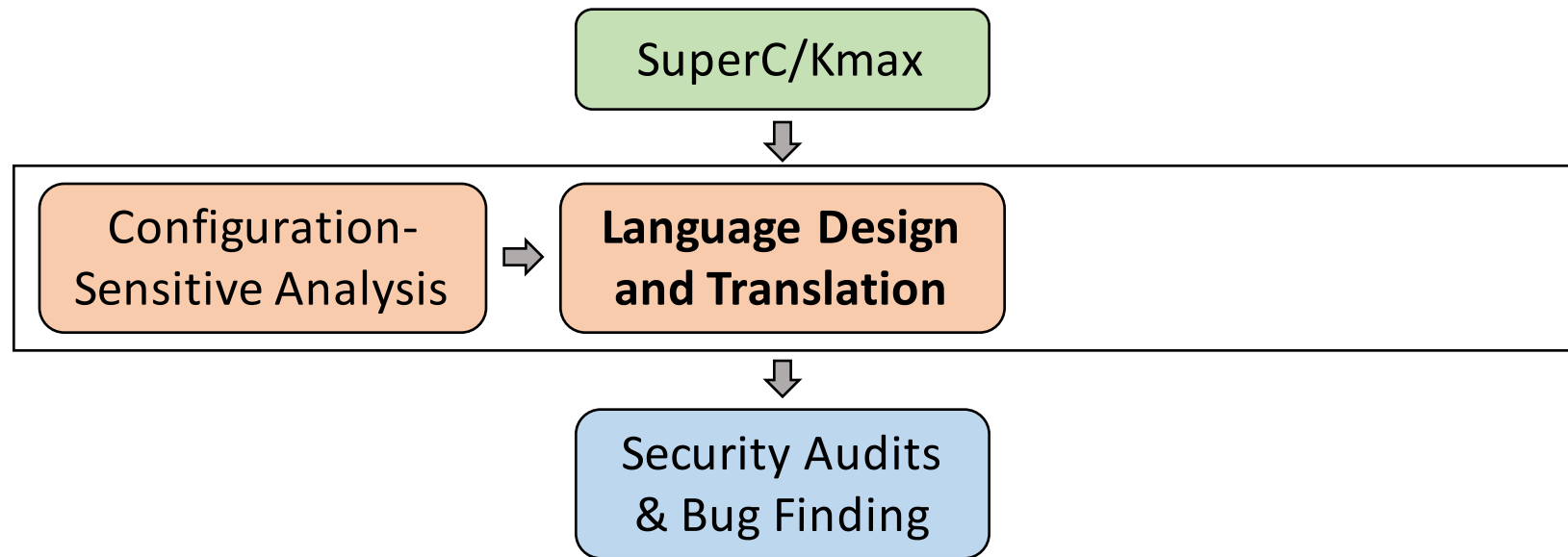
*More reliable and secure
software systems*



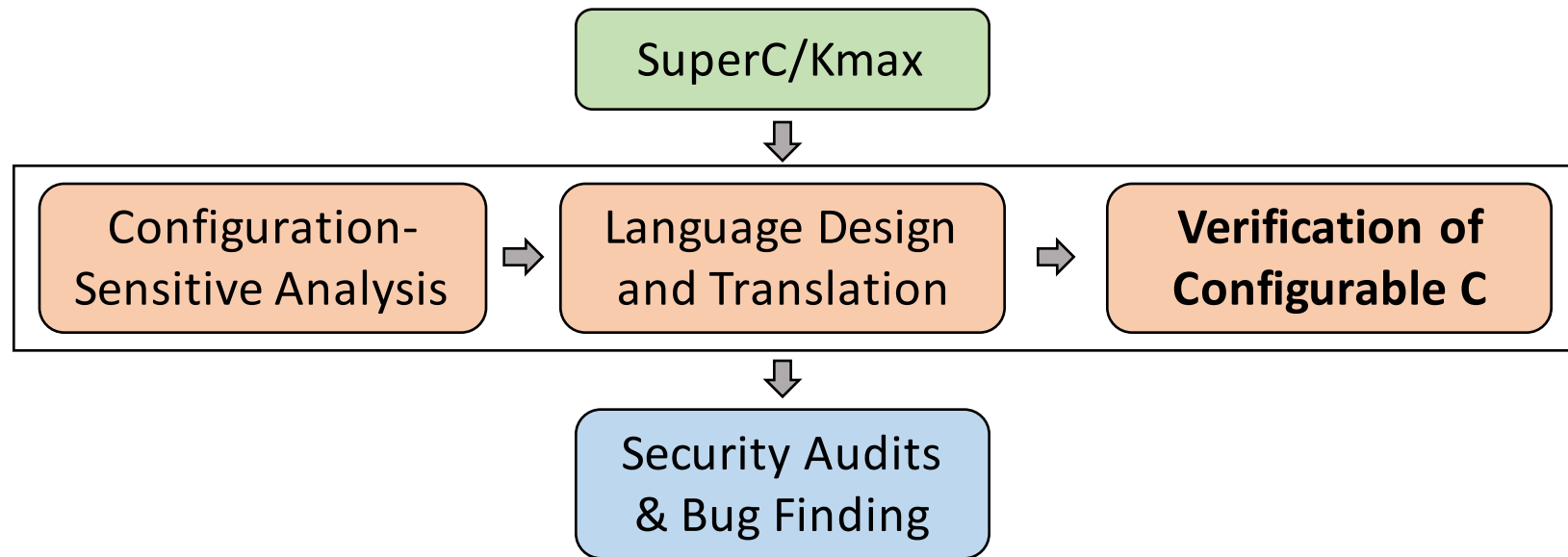
- Static analysis algorithms for configurable software
 - Control flow, callgraph, points-to, information flow
- Challenge: configuration explosion problem
- Future contributions
 - Trade offs between precision and scalability
 - Empirical evaluation on real-world software
 - Higher reliability of systems software



- Collaborations for dynamic analysis
 - Optimizing configurations with Oh, Batory (UT Austin)
 - iGen with Nguyen (UNL), Koc (UMD), Wei (UT Dallas)
 - Off-the-shelf bug-finders with Wei (UT Dallas)



- New C language extensions to replace preprocessor, make
- Challenge: balancing expressivity and ease-of-analysis
- Future contributions
 - Improved development ecosystem for programmers
 - Persuading systems programmers to trust the compiler
- Collaboration opportunities
 - Language designers
 - Machine learning to assist translation



- Developing formal semantics for configurable code
- Challenge: integrating with existing verified C toolchain
- Future contributions:
 - Formal semantics of preprocessor, Makefiles
 - Verified configurable systems
- Collaboration opportunities
 - Formal verification experts

Overview

