Verifying the Safety of Security-Critical Applications

Thomas Dillig
Stanford University
Reliability and security of software is a huge problem.
Reliability and security of software is a huge problem.

I happen to need to call 911 one night, found that my phone crashes every time I dial 911. My wife's phone does not do that, any thought? By the way, it is hard to test this problem due to the sensitivity of calling 911 repeatedly. Thanks,

heartboken
We want to give guarantees about program behavior, for all possible executions and without running the program.
Recall that deciding any non-trivial property about an arbitrary program is undecidable.
Recall that deciding any non-trivial property about an arbitrary program is undecidable.

⇒ Cannot exactly characterize the set of states a program \( P \) may be in.
Recall that deciding any non-trivial property about an arbitrary program is undecidable.

⇒ Cannot exactly characterize the set of states a program $P$ may be in.

Solution: Overapproximate program behavior.
Error states outside over-approximation
⇒ Program verified
Error states outside over-approximation
⇒ Program verified

Error states inside over-approximation, but outside $P$
⇒ false alarm
Undecidability and Abstraction

- Error states outside over-approximation
  \[\Rightarrow\] Program verified

- Error states inside over-approximation, but outside $P$
  \[\Rightarrow\] false alarm

\[\Rightarrow\] Goal: Construct an abstraction precise enough to prove properties about realistic programs
Our Goal in This Talk

- Multiple orthogonal issues in constructing precise abstractions
Our Goal in This Talk

- Multiple orthogonal issues in constructing precise abstractions

- **This talk:** Construct a precise and efficient abstraction that allows automatically proving properties about *container-manipulating* programs
Containers

General-purpose data structures for inserting, retrieving, removing, and iterating over elements
Containers

General-purpose data structures for inserting, retrieving, removing, and iterating over elements

- **Examples:** Array, vector, list, map, set, stack, queue, ...
Containers

General-purpose data structures for inserting, retrieving, removing, and iterating over elements

- **Examples**: Array, vector, list, map, set, stack, queue, …
- Widely used; provided by common programming languages
Containers

General-purpose data structures for inserting, retrieving, removing, and iterating over elements

- **Examples:** Array, vector, list, map, set, stack, queue, ...

- Widely used; provided by common programming languages

⇒ Precise static reasoning about containers crucial for successful verification
Example

Modeled after method `responseDataCallList` in the telephony module in the Android phone code base

```c++
void send_data(vector<Data*> & data) {
    list<Packet*> packets;
    for(int j=0; j < data.size(); j++) {
        Packet* cur = new Packet();
        cur->val = data[j];
        packets.push_front(cur);
    }
    Data* last = data[data.size()-1];
    assert(packets.front()->val == last);
}
```
We want to prove that this program will not crash due to an assertion failure in any execution.

```c++
void send_data(vector<Data*> & data) {
    list<Packet*> packets;
    for(int j=0; j < data.size(); j++) {
        Packet* cur = new Packet();
        cur->val = data[j];
        packets.push_front(cur);
    }
    Data* last = data[data.size()-1];
    assert(packets.front()->val == last);
}
```
Example

```cpp
void send_data(vector<Data*> & data) {
    list<Packet*> packets;
    for (int j = 0; j < data.size(); j++) {
        Packet* cur = new Packet();
        cur->val = data[j];
        packets.push_front(cur);
    }
    Data* last = data[data.size()-1];
    assert(packets.front()->val == last);
}
```
void send_data(vector<Data*> & data) {
    list<Packet*> packets;
    for(int j=0; j < data.size(); j++) {
        Packet* cur = new Packet();
        cur->val = data[j];
        packets.push_front(cur);
    }
    Data* last = data[data.size()-1];
    assert(packets.front()->val == last);
}
void send_data(vector<Data*> & data) {
    list<Packet*> packets;
    for(int j=0; j < data.size(); j++) {
        Packet* cur = new Packet();
        cur->val = data[j];
        packets.push_front(cur);
    }
    Data* last = data[data.size()-1];
    assert(packets.front()->val == last);
}
void send_data(vector<Data*> & data) {
    list<Packet*> packets;
    for(int j=0; j < data.size(); j++) {
        Packet* cur = new Packet();
        cur->val = data[j];
        packets.push_front(cur);
    }
    Data* last = data[data.size()-1];
    assert(packets.front()->val == last);
}
void send_data(vector<Data*> & data) {
    list<Packet*> packets;
    for(int j=0; j < data.size(); j++) {
        Packet* cur = new Packet();
        cur->val = data[j];
        packets.push_front(cur);
    }
    Data* last = data[data.size()-1];
    assert(packets.front()->val == last);
}
Example

```cpp
void send_data(vector<Data*> & data) {
    list<Packet*> packets;
    for(int j=0; j < data.size(); j++) {
        Packet* cur = new Packet();
        cur->val = data[j];
        packets.push_front(cur);
    }

    Data* last = data[data.size()-1];
    assert(packets.front()->val == last);
}
```
Example

```cpp
void send_data(vector<Data*> & data) {
    list<Packet*> packets;
    for(int j=0; j < data.size(); j++) {
        Packet* cur = new Packet();
        cur->val = data[j];
        packets.push_front(cur);
    }
    Data* last = data[data.size()-1];
    assert(packets.front()->val == last);
}
```
Example

- Such assertions are very difficult to prove automatically

```cpp
void send_data(vector<Data*> & data) {
    list<Packet*> packets;
    for(int j=0; j < data.size(); j++) {
        Packet* cur = new Packet();
        cur->val = data[j];
        packets.push_front(cur);
    }
    Data* last = data[data.size()-1];
    assert(packets.front()->val == last);
}
```
Such assertions are very difficult to prove automatically.

**Difficulties:**

```cpp
void send_data(vector<Data*> & data) {
    list<Packet*> packets;
    for(int j=0; j < data.size(); j++) {
        Packet* cur = new Packet();
        cur->val = data[j];
        packets.push_front(cur);
    }
    Data* last = data[data.size()-1];
    assert(packets.front()->val == last);
}
```
Such assertions are very difficult to prove automatically.

**Difficulties:**

- Number of elements in data unknown

```cpp
void send_data(vector<Data*> & data)
{
    list<Packet*> packets;
    for(int j=0; j < data.size(); j++) {
        Packet* cur = new Packet();
        cur->val = data[j];
        packets.push_front(cur);
    }
    Data* last = data[data.size()-1];
    assert(packets.front()->val == last);
}
```
Example

- Such assertions are very difficult to prove automatically

- **Difficulties:**
  - Number of elements in `data` unknown
  - Unknown number of dynamically allocated objects

```cpp
void send_data(vector<Data*> & data) {
    list<Packet*> packets;
    for(int j=0; j < data.size(); j++) {
        Packet* cur = new Packet();
        cur->val = data[j];
        packets.push_front(cur);
    }
    Data* last = data[data.size()-1];
    assert(packets.front()->val == last);
}
```
The Standard Memory Abstraction

- **Standard abstraction**: Represent statically unknown number of memory locations using *summary locations*

  ![Diagram showing packets, alloc, and data with edges connecting them.]

  ```
  assert(packets.front()->val == last)
  ```

  ```
  Edge from node A to B ≡ Any concrete location in A may point to any concrete location in B
  ```

  ```
  ⇒ a full cross-product
  ```
**Standard abstraction:** Represent statically unknown number of memory locations using **summary locations**

- **Standard memory abstraction:**
  - Represent staticaly unknown number of memory locations using summary locations

```
assert(packets.front()->val == last)
```
Standard abstraction: Represent statically unknown number of memory locations using summary locations

Summary location represents multiple run-time locations
The Standard Memory Abstraction

- **Standard abstraction**: Represent statically unknown number of memory locations using **summary locations**

- **Summary location** represents multiple run-time locations

- **Edge from node A to B** $\equiv$ Any concrete location in A may point to any concrete location in B
The Standard Memory Abstraction

- **Standard abstraction**: Represent statically unknown number of memory locations using summary locations

- Summary location represents multiple run-time locations

- **Edge from node A to B**: Any concrete location in A may point to any concrete location in B ⇒ a full cross-product
The Standard Memory Abstraction

- **Standard abstraction**: Represent statically unknown number of memory locations using *summary locations*

- Summary location represents multiple run-time locations

- **Edge from node A to B**: Any concrete location in A may point to any concrete location in B
  \[ \Rightarrow \text{a full cross-product} \]

```
assert(packets.front()->val == last)  
```
The Challenge

- No existing technique can prove such assertions automatically
No existing technique can prove such assertions automatically.

But real programs heavily use containers...
Our Contribution

First analysis that is precise enough to automatically prove such properties about container- and heap-manipulating programs in a scalable way.
Our Approach

Overarching idea:

Symbolic heap abstraction that combines logical formulae with a graph representation to describe contents of containers
Indexed Locations

Key Idea 1:
Represent containers with indexed locations
Indexed Locations

Key Idea 1:

Represent containers with indexed locations

- list<Packet*> data represented using a single abstract location qualified by index variable
Indexed Locations

Key Idea 1:
Represent containers with indexed locations

- list<Packet*> data represented using a single abstract location qualified by index variable
- Index variable ranges over possible indices of list
Indexed Locations

Key Idea 1: Represent containers with indexed locations

- `list<Packet*> data` represented using a single abstract location qualified by index variable
- Index variable ranges over possible indices of list
Indexed Locations

Key Idea 1:

Represent containers with indexed locations

- list<Packet*> data represented using a single abstract location qualified by index variable
- Index variable ranges over possible indices of list
void send_data(vector<Data*> & data) {
    list<Packet*> packets;
    for(int j=0; j < data.size(); j++) {
        Packet* cur = new Packet();
        cur->val = data[j];
        packets.push_front(cur);
    }
    Data* last = data[data.size()-1];
    assert(packets.front()->val == last);
}
**Example**

```cpp
void send_data(vector<Data*> & data) {
    list<Packet*> packets;
    for(int j=0; j < data.size(); j++) {
        Packet* cur = new Packet();
        cur->val = data[j];
        packets.push_front(cur);
    }
    Data* last = data[data.size()-1];
    assert(packets.front()->val == last);
}
```

- **Statically unknown number of allocations in the loop**
Example

void send_data(vector<Data*> & data)
{
    list<Packet*> packets;
    for(int j=0; j < data.size(); j++) {
        Packet* cur = new Packet();
        cur->val = data[j];
        packets.push_front(cur);
    }
    Data* last = data[data.size()-1];
    assert(packets.front()->val == last);
}

- Statically unknown number of allocations in the loop
- $\langle \alpha \rangle_{i_2}$ represents all allocations
void send_data(vector<Data*> & data) {
    list<Packet*> packets;
    for(int j=0; j < data.size(); j++) {
        Packet* cur = new Packet();
        cur->val = data[j];
        packets.push_front(cur);
    }
    Data* last = data[data.size()-1];
    assert(packets.front()->val == last);
}

- Statically unknown number of allocations in the loop
- \(\langle \alpha \rangle_{i_2}\) represents all allocations
- For instance, \(\langle \alpha \rangle_0\) represents allocation in the first iteration
Edge Constraints

Key Idea 2:
Points-to edges are qualified by constraints on index variables.
Key Idea 2:
Points-to edges are qualified by constraints on index variables.
- Consider pointer relations between containers $a$ and $b$. 
Edge Constraints

Key Idea 2:
Points-to edges are qualified by constraints on index variables.

- Consider pointer relations between containers \( a \) and \( b \).
Key Idea 2:

Points-to edges are qualified by constraints on index variables.

- Consider pointer relations between containers $a$ and $b$.

$\Rightarrow$ Constraints on points-to edges allow richer relations than cross-product
void send_data(vector<Data*> & data) {
    list<Packet*> packets;
    for(int j=0; j < data.size(); j++) {
        Packet* cur = new Packet();
        cur->val = data[j];
        packets.push_front(cur);
    }
    Data* last = data[data.size()-1];
    assert(packets.front()->val == last);
}

\[ \langle \text{packets} \rangle_{i_1} \]
\[ 0 \leq i_1 < \text{size(data)} \land i_2 = \text{size(data)} - i_1 - 1 \]

\[ \langle \alpha \rangle_{i_2} \]
\[ i_3 = i_2 \]

\[ \langle \text{data} \rangle_{i_3} \]
```c++
void send_data(vector<Data*> & data) {
    list<Packet*> packets;
    for (int j = 0; j < data.size(); j++) {
        Packet* cur = new Packet();
        cur->val = data[j];
        packets.push_front(cur);
    }
    Data* last = data[data.size()-1];
    assert(packets.front()->val == last);
}
```
Example

```
void send_data(vector<Data*> & data)
{
    list<Packet*> packets;
    for(int j=0; j < data.size(); j++) {
        Packet* cur = new Packet();
        cur->val = data[j];
        packets.push_front(cur);
    }
    Data* last = data[data.size()-1];
    assert(packets.front()->val == last);
}
```
We saw how to **represent** points-to relations in our symbolic heap representation.
We saw how to **represent** points-to relations in our symbolic heap representation.

How do we **use** the information encoded in this abstraction, e.g., to prove the assertion:

```c++
assert(packets.front()->val == last);
```
Load Example

Consider `packets.front()->val`

\[
\langle \text{packets} \rangle_{i_1} \\
0 \leq i_1 < \text{size}(data) \land \\
i_2 = \text{size}(data) - i_1 - 1 \\
i_3 = i_2 \\
\langle \alpha \rangle_{i_2} \rightarrow \langle \text{data} \rangle_{i_3}
\]
Load Example

Consider `packets.front()->val`

- Determine where front of `packets` points to \( \Rightarrow i_1 = 0 \)
Consider \texttt{packets.front()->val}

- Determine where front of \texttt{packets} points to $\Rightarrow i_1 = 0$
- What is the value of $i_2$?

\[
\begin{align*}
\langle \text{packets}\rangle_{i_1} & \quad i_1 = 0 \\
\langle \alpha\rangle_{i_2} & \quad 0 \leq i_1 < \text{size(data)} \land i_2 = \text{size(data)} - i_1 - 1 \\
\langle \text{data}\rangle_{i_3} & \quad i_3 = i_2
\end{align*}
\]
Load Example

Consider \texttt{packets.front() \rightarrow val}

- Determine where front of \texttt{packets} points to \( \Rightarrow i_1 = 0 \)
- What is the value of \( i_2 \)?
  \[
i_2 = \text{size(data)} - i_1 - 1 \land 0 \leq i_1 < \text{size(data)}
\]
Consider `packets.front()->val`.

- Determine where front of packets points to $⇒ i_1 = 0$

- What is the value of $i_2$?

$\exists i_1. i_1 = 0 ∧ i_2 = \text{size(data)} - i_1 - 1$

$∧ 0 ≤ i_1 < \text{size(data)}$
Consider `packets.front()->val`

- Determine where front of `packets` points to \( \Rightarrow i_1 = 0 \)
- What is the value of \( i_2 \)?

\[
i_2 = \text{size(data)} - 1
\]
Consider `packets.front()->val`

- Determine where front of packets points to $\Rightarrow i_1 = 0$
- What is the value of $i_2$?
  $$i_2 = size(data) - 1$$

Can now prove assertion `packets.front()->val == last`
Indexed locations + constraints on index variables = Precise per-element container reasoning
Advantages of Symbolic Heap

Indexed locations + constraints on index variables = Precise per-element container reasoning

• Employ machinery from standard logic to:
Advantages of Symbolic Heap

Indexed locations + constraints on index variables =
Precise per-element container reasoning

Employ machinery from standard logic to:
1. traverse heap references: satisfiability, existential quantifier elimination
Advantages of Symbolic Heap

Indexed locations + constraints on index variables = Precise per-element container reasoning

- Employ machinery from standard logic to:
  1. traverse heap references: satisfiability, existential quantifier elimination
  2. analyze updates precisely and uniformly: negation, conjunction
Indexed locations + constraints on index variables = Precise per-element container reasoning

- Employ machinery from standard logic to:
  1. traverse heap references: satisfiability, existential quantifier elimination
  2. analyze updates precisely and uniformly: negation, conjunction

⇒ Reduce container reasoning to integer constraints and standard logic operations
Implemented heap/container analysis in our \texttt{Compass} program analysis framework for C and C++ programs.
Implementation

- Implemented heap/container analysis in our Compass program analysis framework for C and C++ programs.

- Analysis requires solving constraints in combined theory of linear integer inequalities and uninterpreted functions

  ⇒ used our Mistral SMT solver.
Experiments

- Analyzed real open-source C and C++ applications using containers
Experiments

- Analyzed real open-source C and C++ applications using containers
  - LiteSQL, 16,030 LOC
Experiments

- Analyzed real open-source C and C++ applications using containers
  - LiteSQL, 16,030 LOC
  - OpenSSH, 26,615 LOC
Experiments

- Analyzed real open-source C and C++ applications using containers
  - LiteSQL, 16,030 LOC
  - OpenSSH, 26,615 LOC
  - Inkscape Widget Library, 37,211 LOC
Analyzed real open-source C and C++ applications using containers

- LiteSQL, 16,030 LOC
- OpenSSH, 26,615 LOC
- Inkscape Widget Library, 37,211 LOC
- DigiKam, 128,318 LOC
Experiments

- Analyzed real open-source C and C++ applications using containers
  - LiteSQL, 16,030 LOC
  - OpenSSH, 26,615 LOC
  - Inkscape Widget Library, 37,211 LOC
  - DigiKam, 128,318 LOC
- Annotated containers provided by the STL and QT libraries
Application: Memory Safety

- Ran our verification tool to find all segmentation faults or run-time exceptions caused by:
  - buffer overruns and underruns
  - null dereference errors
  - accessing deleted memory
- Also checked memory leaks

```c
switch (iFilterType)
{
    case CN_FILTERBYPASS:
    case CN_FILTERB/HIGHPASS:
        hrng[2] = CreateEllipticRgn(x11, y11, x12, y12);
        break;
    case CN_FILTERBLOWPASS:
        hrng[0] = CreateEllipticRgn(x11, y11, x12, y12);
        hrng[1] = CreateEllipticRgn(x13, y13, x14, y14);
        hrng[2] = CreateEllipticRgn(0, 0, 10, 10);
        CombineRgn(&hrng[2], hrng[0], hrng[1], RGN_XOR);
        DeleteObject(hrng[0]);
        DeleteObject(hrng[1]);
        break;
    case CN_FILTERHANDB:
        hrng[0] = CreateEllipticRgn(0, 0, 20, 20);
        hrng[1] = CreateEllipticRgn(0, 0, 10, 10);
        hrng[2] = CreateEllipticRgn(0, 0, 5, 5);
        DeleteObject(hrng[0]);
        DeleteObject(hrng[2]);
        break;
    case CN_FILTERH/BLOWPASS:
        hrng[0] = CreateEllipticRgn(0, 0, 20, 20);
        hrng[1] = CreateEllipticRgn(0, 0, 10, 10);
        hrng[2] = CreateEllipticRgn(0, 0, 5, 5);
        CombineRgn(hrng[0], &hrng[2], &hrng[1], RGN_XOR);
        DeleteObject(hrng[0]);
        DeleteObject(hrng[1]);
        break;
    case CN_FILTERL:
        // Tegn det endelige canvas i rød
        FillRgn(Fastintinfo.hdc, hrng[3], hbrush);
        // Fjern de allokerte regionene, de er bare midlertidige
        for(i=0;i<4;i++)
            if (hrng[i]!=NULL)
                DeleteObject(hrng[i]);
```
First Experiment:

- Represent containers as bags of values using the standard abstraction
First Experiment:

- Represent containers as bags of values using the standard abstraction

- Existing tools that analyze programs of this size use this abstraction
First Experiment:
- Represent containers as bags of values using the standard abstraction

- Existing tools that analyze programs of this size use this abstraction

⇒ Cannot track index-to-value correlations, modification to one container element contaminates all others
Containers as Bags

False alarms for standard abstraction

Bugs found

LiteSQL OpenSSH Inkscape DigiKam

Treating containers as bags leads to unacceptable number of false alarms.
Conclusion

Treating containers as bags leads to unacceptable number of false alarms.
Second Experiment:

- Used the techniques described in this talk: indexed locations, symbolic points-to relations
Second Experiment:

- Used the techniques described in this talk: indexed locations, symbolic points-to relations

⇒ Able to track key-value correlations; precise reasoning about heap objects stored in containers
Containers Modeled as Indexed Locations

- False alarms for standard abstraction
- Bugs found
- False alarms for our abstraction

- LiteSQL
- OpenSSH
- Inkscape
- DigiKam

- Huge reduction in number of false alarms
Containers Modeled as Indexed Locations

- False alarms for standard abstraction
- Bugs found
- False alarms for our abstraction

✓ Huge reduction in number of false alarms
Containers Modeled as Indexed Locations

- Bugs found
- False alarms for our abstraction

✓ Analysis reports very few false positives
Containers Modeled as Indexed Locations

Bugs found
False alarms for our abstraction

Cost of the analysis is tractable
Sound, precise, and automatic technique for verifying real programs that make use of heap allocations and containers
Summary

- Sound, precise, and automatic technique for verifying real programs that make use of heap allocations and containers
- First technique to verify real programs of this size with very few false alarms
Summary

- Sound, precise, and automatic technique for verifying real programs that make use of heap allocations and containers
- First technique to verify real programs of this size with very few false alarms
- Substantial improvement over the state-of-the art ⇒ substantially extends class of programs that can be automatically verified
Related Work


