Verifying Real-world Cryptography

Mike Dodds

March 7, 2019
University of Washington
Who Galois are

Research and development lab of ~100 people

Locations

Portland, OR

Arlington, VA

Dayton, OH
What Galois do

*Formal methods research meets real-world applications*

Programming languages, analysis, verification, security, cryptography

Our tools:

- Symbolic execution
- Model checking
- Interactive theorem provers
- Functional programming (esp. Haskell)
Galois clients

Big research projects from US govt

Commercial research projects e.g. Amazon, Facebook, others

Lots of collaborations with academic partners
Galois formal methods priorities

- Tools for real languages / systems (vs proofs of concept)
- Highly automated tools (vs manual proofs)
- Domain-specific tools / languages (vs universal tools)
- Increasing system assurance (vs absolute correctness)
- Integrating with SWE workflows (vs demanding changes)
Formal Methods for Security
Computer Security Imbalance

1. Defenders have to prevent all problems.
2. Attackers need only find one entry point.

- Verification aims to enable #1 for critical core components
- Works on small code bases. Useful in practice because
  - Many systems are engineered to keep the security-critical core small (hypervisors, OS kernels, secure channels).
  - Technology advances have decreased the overall level of effort.
A Grid of Bugs
A Grid of Bugs

Program States
A Grid of Bugs

- Granting access to authorized user
- Processing a transaction

Program States
A Grid of Bugs

- Granting access to authorized user
- Processing a transaction
- Granting access to unauthorized user

Program States
Software Security As A Game

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Defender’s Turn: Pick 10 Squares
Defender’s Turn: Fix Problems
Defender’s Turn: Fix Problems
Attacker’s Turn: Pick 10 (or 20, or…)

[Diagram of a grid with blue squares]
Attacker’s Turn: Pick 10 (or 20, or...)

![Grid Diagram]
Attacker Advantage

• General per-round odds favor attacker.
  • Find all red squares vs. find any red square

• Attacker generally has more time.
  • Windows XP is 15 years old now.
Verification / Formal Methods

Cover *much* more of the state space by discovering and leveraging underlying *structure*. 
Formal Methods: Characterize State

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 \\
24 & 25 & 26 & 27 & 28 & 29 & 30 & 31 \\
32 & 33 & 34 & 35 & 36 & 37 & 38 & 39 \\
40 & 41 & 42 & 43 & 44 & 45 & 46 & 47 \\
48 & 49 & 50 & 51 & 52 & 53 & 54 & 55 \\
56 & 57 & 58 & 59 & 60 & 61 & 62 & 63 \\
\end{array}
\]

- \( i \mod 13 = 0 \)
- \( i \mod 8 = i \div 8 \)
Formal Methods: Characterize State

$$i \mod 13 \neq 0$$

$$i \mod 8 \neq i \div 8$$
Formal Methods: Characterize State

\[
i \mod 13 \neq 0
\]

\[
i \mod 8 \neq i \div 8
\]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>48</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>48</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td>56</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td></td>
</tr>
</tbody>
</table>
Formal Methods: Characterize State

\[
\begin{array}{cccccccccccccccccccccccccccccc}
\hline
\text{i mod 13} & \neq & 0 \\
\text{i mod 8} & \neq & \text{i div 8} \\
\hline
\end{array}
\]
Formal Methods

Cover *much* more of the state space by discovering and leveraging underlying *structure*.

In the limit, can prove that code is correct in *all* cases (for the given proof scope).

Method: Characterize the “good” behavior. Show this is the only behavior that can occur.

This is now at a viable cost / benefit point for critical, broadly deployed code.
Galois tools: Cryptol and SAW
Specification language: Cryptol

A single, high-level specification for (cryptographic) algorithms

- Cryptol goals
  - Appropriate for cryptography
  - Natural
  - Concise
  - Similar to existing notation
  - Appropriate for execution and verification

- Language features
  - Statically-typed functional language
  - Sized bit vectors (type level naturals)
  - Stream comprehensions (stream diagrams)
A Taste of Cryptol

- Functions and sequences are key notions
- Both can be recursive
- To compute the sequence of all natural numbers

\[ \text{nats} = [0] \# [n + 1 | n \leftarrow \text{nats}] \]
Specifications as code

Cryptol specifications are (Haskell-like) code:

\[
\text{hmac } h \ h2 \ h3 \ K \ m = \\
\quad h2 \ (\text{okey } \# \text{ split } (h \ (\text{ikey } \# m))) \\
\text{where} \\
\quad k0 = kinit \ h3 \ K \\
\quad \text{okey} = [kb \ ^ {0x5C} | kb <- k0] \\
\quad \text{ikey} = [kb \ ^ {0x36} | kb <- k0]
\]

You can use a Cryptol specification in many ways:

• Execute with an input
• Generate input values
• Compile into code.
Verification tool: SAW

- SAW = Software Analysis Workbench
  - Software: many languages
  - Analysis: many types of analysis, focused on functionality
  - Workbench: flexible interface, supporting many goals

- Intended as a flexible tool for software analysis
- What separates it from other systems?
  - One view: compiler :: imperative code → functional code
  - Captures all functional behavior, simplifying later if necessary
  - Uses efficient internal representations tuned to equivalence checking
  - Strong bit vector reasoning support
  - Focus on practicality over novelty

- Open source (BSD3) and available now
SAW verification process

Inputs:
1. Executable specification in Cryptol
2. Target program in C / Java …

Symbolically execute both programs to generate SAWcore terms, a pure intermediate language

Check programs are equivalent, e.g using SMT solvers
SAW architecture
Compositional Verification

Due to its size and complexity, no tool that we know of can verify top-level cryptographic primitives in one go.

To scale up automated tools to larger problems, we need tools for decomposing larger problems into smaller pieces that can be verified individually.

In SAW, we do this by allowing users to verify individual methods independently, and composing the results together in a larger verification effort.

Once a specification is defined, it can be used to simplify later methods.
Example: s2n TLS verification

Correctness of core components in Amazon’s s2n TLS library.
TLS: Transport Layer Security

TLS (newer version of SSL) provides us most of the

Confidentiality
Data-Integrity
Authentication

guarantees that we enjoy on the internet today.
If I go to gmail...

TLS lets me be sure:

1. I’m actually talking to google
2. Nobody (not even my ISP) can read what I’m reading
3. Nobody (not even my ISP) can change the data I’m reading

Also used pervasively for communication between services in the cloud.
Amazon s2n: A TLS Implementation

• Inspired by TLS vulnerabilities discovered by researchers in other implementations.
• Written with security and performance as primary goals.
• Drops some arguably insecure/less secure features.
  • Result: Much smaller, clearer, more auditable code.
  • OpenSSL TLS is 70k lines of C code.
  • s2n is only 6k.
• Used in production at Amazon.
HMAC: A Component of TLS

- keyed-Hash Message Authentication Code
- Provides a signature for a message that confirms:
  - Authenticity: the message was signed by the expected sender
  - Integrity: the message has not been modified

\[
\text{HMAC}(K, m) = H((K0 \oplus \text{opad}) \| H((K0 \oplus \text{ipad}) \| m))
\]

- Code is still complex
  - 521 lines of C code
HMAC Specification

C HMAC

HMAC(m, k) =
H((K ⊕ opad) || H((K ⊕ ipad) || m))

Concise

Easily auditable

Interoperable

Fast

Goal: bridge this gap
Summary of Approach

1. Write the formal specification.
2. Write some “scaffolding” to bridge the gap between specification and C code.
3. Apply automated tools.
4. Integrate into development environment.

About 2 months of effort.
Summary of Approach

1. Write the formal specification.
2. Write some “scaffolding” to bridge the gap between specification and C code.
3. Apply automated tools.
4. Integrate into development environment.

About 2 months of effort.
Summary of Approach

HMAC(K, m) = H((K0 ⊕ opad) ⊕ H((K0 ⊕ ipad) ⊕ m))

Step 1: Capture this specification in a formal language (we used Cryptol)

hmac h h2 h3 K m =
    h2 (okey # split (h (ikey # m)))
where
    k0 = kinit h3 K
    okey = [kb ^ 0x5C | kb <- k0]
    ikey = [kb ^ 0x36 | kb <- k0]
HMAC Specification

hmac h h2 h3 K m =

h2 (okey ≠ split (h (ikey ≠ m)))
where
k0 = kinit h3 K
okey = [kb ^ 0x5C | kb <- k0]
ikey = [kb ^ 0x36 | kb <- k0]

C HMAC

Goal: bridge this gap

Easy auditable

Concise

Fast

Interoperable
Summary of Approach

1. Write the formal specification.
2. Write some “scaffolding” to bridge the gap between specification and C code.
3. Apply automated tools.
4. Integrate into development environment.

About 2 months of effort.
Bridging the gap

Solution: Layers of Abstraction

High-level Cryptol Code

Lower-level Cryptol Code

Production s2n code

Incorporates:
- s2n Data structures
- s2n API

Omits:
- Pointers and memory allocation
- Low-level performance optimization

Increasingly implementation-specific

Specification from before

Unmodified code from s2n repo
Summary of Approach

1. Write the formal specification.
2. Write some “scaffolding” to bridge the gap between Specification and C code.
3. Apply automated tools.
4. Integrate into development environment.

About 2 months of effort.
Bridging the gap

Solution: Layers of Abstraction

Increasingly implementation-specific

High-level Cryptol Code

Lower-level Cryptol Code

Production s2n code

Automatically Constructed by SAW (Software Analysis Workbench) via translation to SMT and application of constraint solvers
Verified HMAC pipeline

(From our CAV18 paper)
Summary of Approach

1. Write the formal specification.
2. Write some “scaffolding” to bridge the gap between Specification and C code.
3. Apply automated tools.
4. Integrate into development environment.

About 2 months of effort.
Continuous Integration

- Proofs run automatically on code changes
  - Proof failure is a build failure
- Proof is independent of exact C code, depends only on:
  - Interfaces (arguments and struct layouts)
  - Function call structure
- Proof is easily adapted:
  - Function body changes $\rightarrow$ likely no proof changes
  - Interface changes $\rightarrow$ similarly-sized proof changes
  - Call structure changes $\rightarrow$ tiny proof changes
Travis CI

awslabs / s2n  build passing

Current  Branches  Build History  Pull Requests  Build #953

master  Merge pull request #517 from xonatius/allocatorOverrides

- Added guards around allocatorOverrides

  - Commit 02ade5e
  - Compare 9eb9b99..02ade5e
  - Branch master

  - Matthew Baldwin authored
  - GitHub committed

Build Jobs

- # 953.1  Xcode: xcode8 C  TESTS=ctverif  4 min 59 sec
Proof Metrics

We have run 12 builds on branch demo and verified 1629 properties. To gain equivalent assurance through test cases we would need to run \(2.4\times10^{130}\) tests.

Run Summary:

For all builds we ran 1629 verifications.

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Build</th>
<th>Size</th>
<th>Equivalent Tests</th>
<th>Complexity*</th>
<th>Time</th>
<th>Succ</th>
</tr>
</thead>
<tbody>
<tr>
<td>hmac_c_state_correct size 0</td>
<td></td>
<td>109</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>hmac_c_state_correct size 1</td>
<td></td>
<td>109</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>hmac_c_state_correct size 128</td>
<td></td>
<td>109</td>
<td>128</td>
<td>3.40e38</td>
<td>0</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>MD5, key size = 64, msg size = 1</td>
<td>s2n_hmac_update</td>
<td>109</td>
<td>65</td>
<td>3.68e19</td>
<td>56,665</td>
<td>3.451</td>
<td>✓</td>
</tr>
<tr>
<td>MD5, key size = 64, msg size = 1</td>
<td>s2n_hmac_digest</td>
<td>109</td>
<td>65</td>
<td>3.68e19</td>
<td>2,972</td>
<td>0.659</td>
<td>✓</td>
</tr>
</tbody>
</table>

Number of SAW verifications this week : 0
Number of SAW verifications all time : 1629
Number of failed SAW checks this week : 0
Number of failed SAW checks all time : 0
Average successful SAW runtime this week : No recorded times
Average successful SAW runtime all time : 2.1 seconds
Average LOC covered by SAW reasoning this week : 103
Average LOC covered by SAW reasoning all time : 103
Average LOC of specification covered by SAW reasoning this week : 787
Protocol Correctness

- Correct implementation of authentication and key exchange.
Protocol Correctness

- Previous work targeted correctness of underlying crypto.
- The protocol level is also security critical (and sometimes wrong)
Protocol Problem Example

• Early ChangeCipherSpec (Early CCS)

“If a ChangeCipherSpec message is injected into the connection after the ServerHello, but before the master secret has been generated, then ssl3_do_change_cipher_spec will generate the keys (2) and the expected Finished hash (3) for the handshake with an empty master secret. This means that both are based only on public information.”

From https://www.imperialviolet.org/2014/06/05/earlyccs.html
State Machine Attacks

- Unexpected message ordering in other implementations causes authentication steps to be bypassed.

From https://www.mitls.org/pages/attacks/SMACK
Protocol State Machine

Goal: bridge this gap
Fixing the problem

- Write a model of the state machine in Cryptol.
- Verify equivalence using SAW
- Integrate into CI
Other Crypto Work

- Have verified implementations of AES, SHA, ECDSA

NISTCurve.java (line 964):

```java
int d = (z[0] & LONG_MASK) + of;
z[0] = (int) d; d >>= 32;
d = (z[1] & LONG_MASK) - of;
z[1] = (int) d; d >>= 32;
d += (z[2] & LONG_MASK);
z[2] = (int) d; d >>= 32;
d += (z[3] & LONG_MASK) + of;
```
Other Crypto Work

- Have verified implementations of AES, SHA, ECDSA

NISTCurve.java (line 964):

```java
    d = (z[0] & LONG_MASK) + of;
    z[0] = (int) d; d >>= 32;
    d += (z[1] & LONG_MASK) - of;
    z[1] = (int) d; d >>= 32;
    d += (z[2] & LONG_MASK);
    z[2] = (int) d; d >>= 32;
    d += (z[3] & LONG_MASK) + of;
```

Bug only occurs when this addition overflows. (rare since of < 5)

SAW found bug in 20 seconds. Testing found bug after 2 hours (8 billion field reductions later).
Other Crypto Work

• Other SAW / Cryptol verification projects:
  • DRBG: Deterministic Random Bit Generator: The main source of cryptographic randomness (see paper at CAV’18)
  • Synthesizing verified hardware crypto
  • Other AWS projects I’m not able to talk about yet
• Working on verifying Facebook Fizz TLS1.3 library
Idea: Continuous Static Analysis
Code quality strategies in industry

Revision Control (git, hg)

Testing (junit, Travis)

Peer Review (GitHub Pull Requests)
Code quality strategies in industry

Revision Control (git, hg)

Testing (junit, Travis)

Peer Review (GitHub Pull Requests)

Static Analysis / verification
Problems with static tools

- False positives, or uninteresting bugs:
  “This part of the code has been well-tested”

- Too many bugs reported:
  “We can’t work through 1000 bug reports”

- Bugs reported too late:
  “We already caught this bug through QA”

- Tools don’t scale to industry projects
Continuous analysis / verification

- *Idea developed in FB, Google, Amazon*

- Run analysis tools when code changes
- Integrate into compilation or continuous integration
- Report results immediately or at code review

- Advantages:
  - Reduce defects that need to be caught in QA / production
  - Improve false positive rate – new code is most likely to be buggy
  - Report bugs to code reviewers – it’s their job to care about bugs
  - Enable scalability

- “*Move fast and don’t break things*”
2019: continuous analysis used by

**Google**
- 1 billion LOC code base
- 20,000 code reviews per day
- Approach: AST patterns
- Example tool: ErrorProne

**Facebook**
- Static analysis of every diff
- Millions of LOC
- Approach: separation logic, abstract interp.
- Example tool: Infer

**Amazon**
- Proofs of correctness
- Core infrastructure
- (millions reqs. per sec.)
- Example tool: SAW
Summary

- Can prove correct behavior rather than search for errors

- For crypto / authentication / access control:

  Behavioral bugs are security bugs

- Proof can be integrated into development workflow to consistently prevent introduction of errors