Self-encrypting Code to Protect against Analysis and Tampering

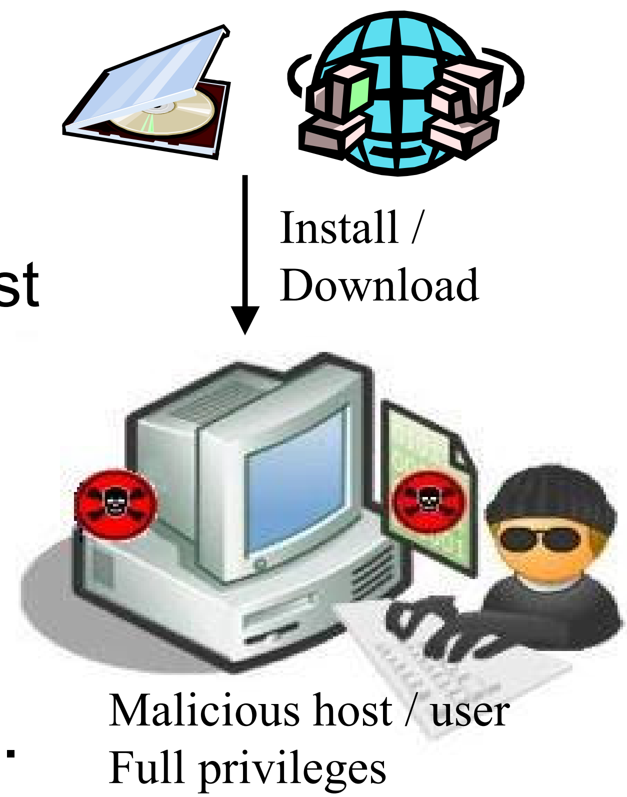
Jan Cappaert
K.U.Leuven/ESAT/COSIC

September 25th, 2007

Contents

Software Protection

- Installing software on a client
  - the owner *looses all control*
  - the software has to *protect itself* against the possibly malicious host (software and user)
- *Software protection* is a collection of all techniques that protect software applications against analysis and tampering.



Software Protection

- An *attack* typically consists of 2 phases:
  1. Analysis
  2. Tampering
- An example:
  1. A company can extract an algorithm, implemented by a competitor, steal it and use it in its own application.
  2. A malicious user modifies the expiration procedure of a software application so he can use it for an extended period of time.

Software Protection

- Software protection techniques
  1. Against *analysis*:
     - Code obfuscation
     - White-box cryptography
     - …
     - Code encryption
  2. Against *tampering*:
     - Code verification
     - …

Software Protection

Technique	Protection against			
	Analysis		Tampering	
	Static	Dyn.	Static	Dyn.
Verification	N	N	F	P
Encryption	F	N	F	N
Obfuscation	P	P	P	P
WB crypto	F	P	F	F

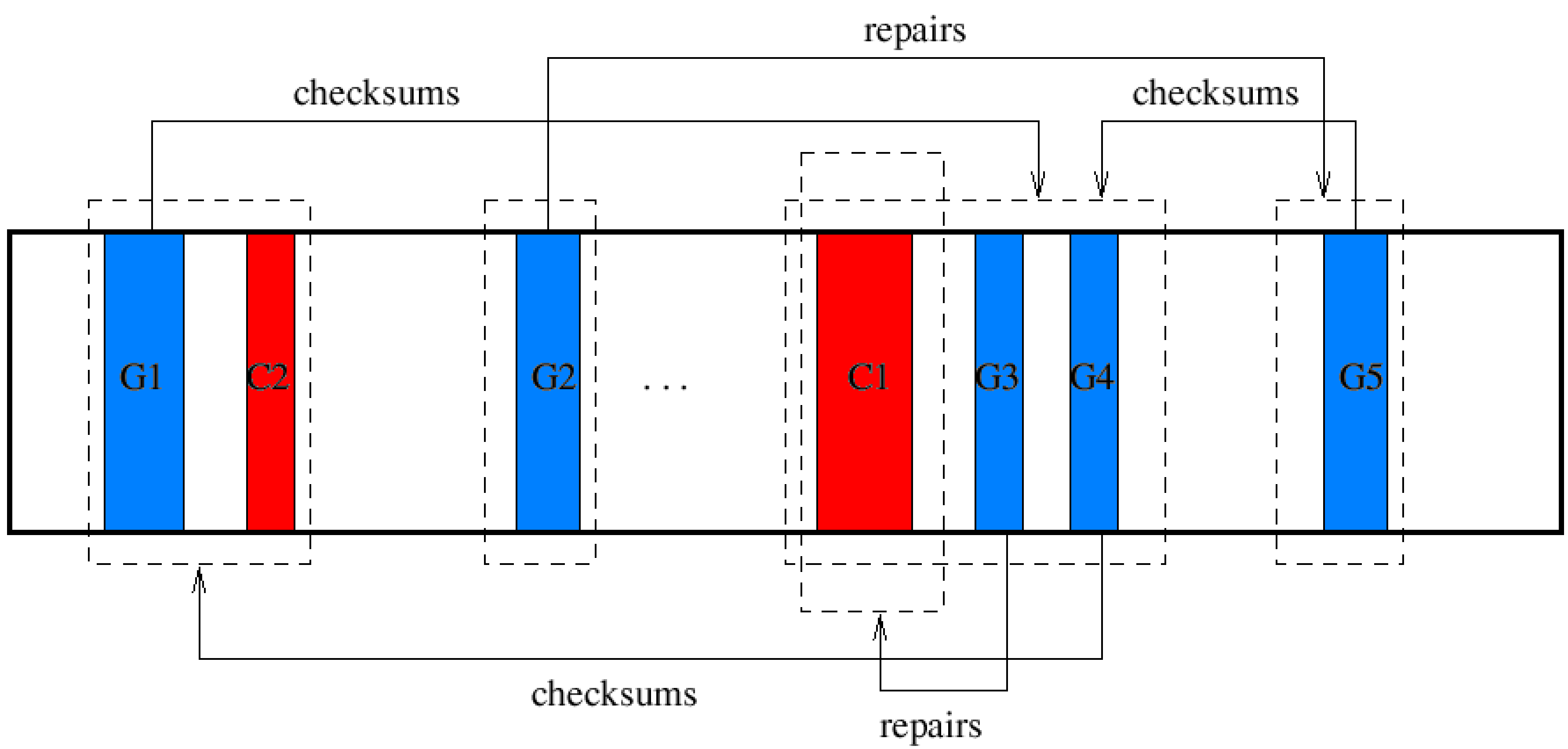
N = None P = Partial F = Full

State of the Art

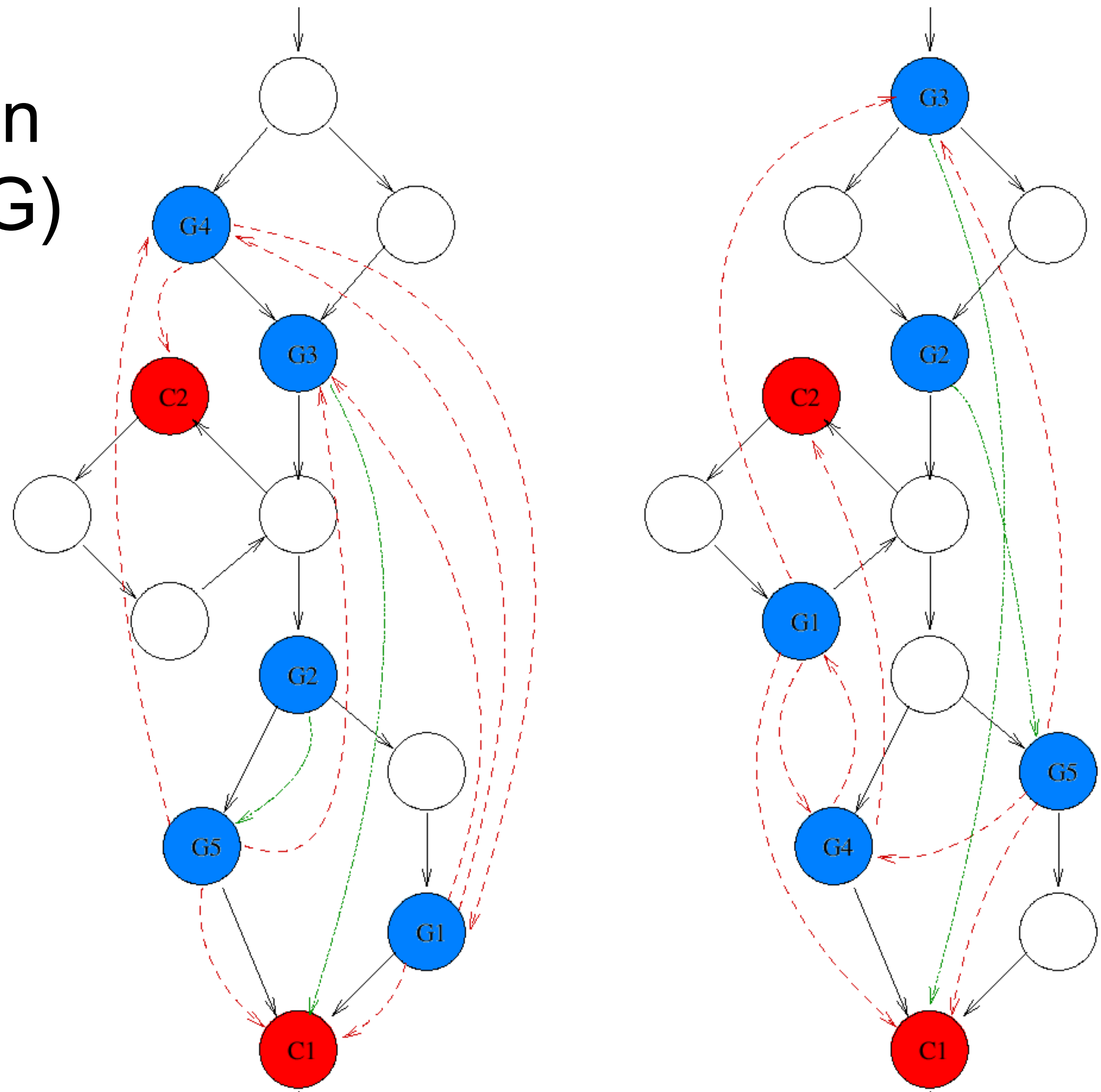
- *“Software guards”* by Chang and Atallah, DRM’01
- *“Testers”* by Horne *et al.* ‘01
- *“Integrity-based encryption”* by Lee *et al.,*’04
- …

Software Guards

- Memory layout:

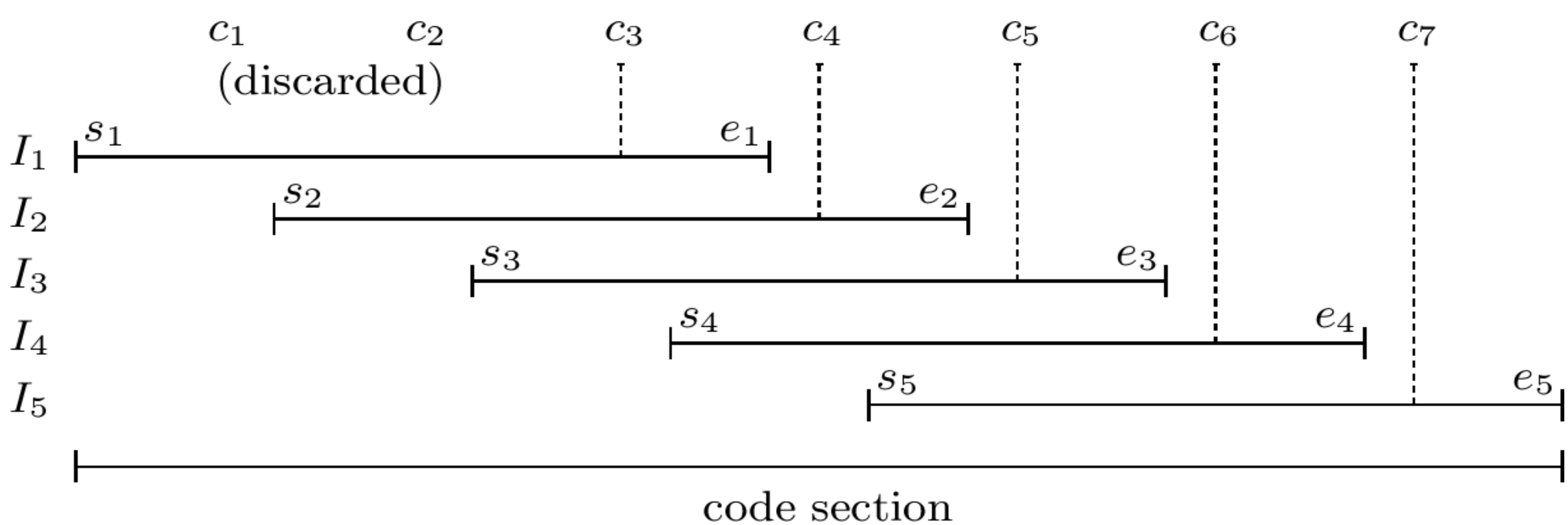


- Execution (see CFG)



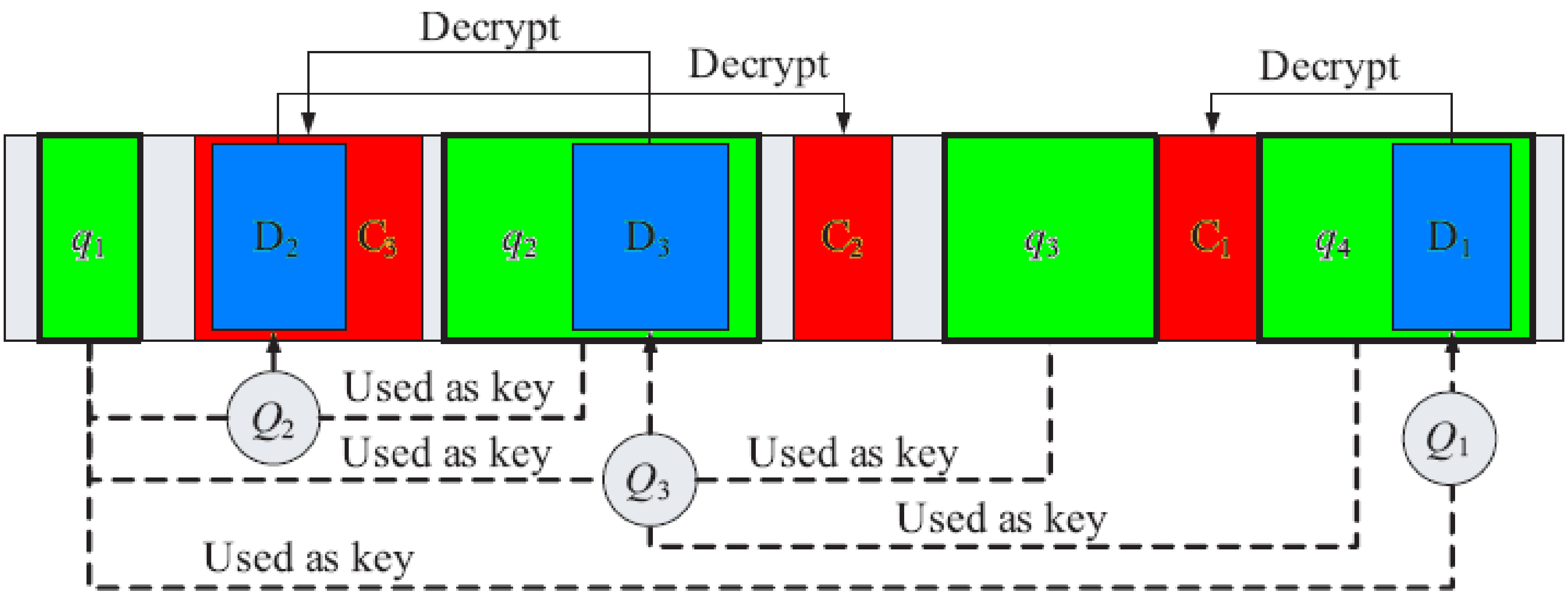
Software guards

- *Testers* and *Correctors*
  - Reversible hash function
  - Watermark



Integrity-based Encryption

- Memory layout:



Analysis and Tamper Resistance

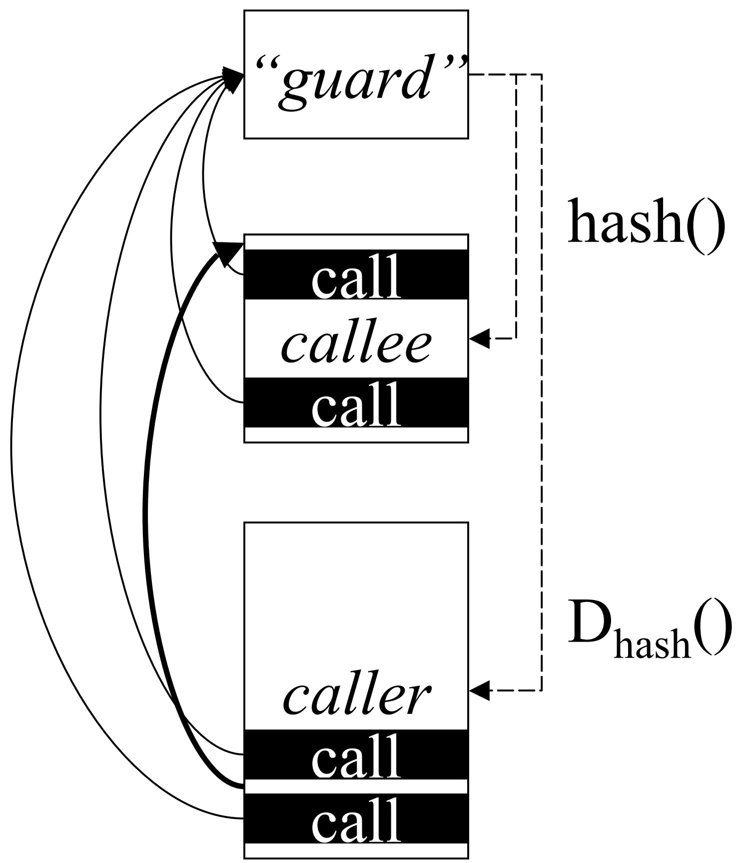
Technique	Protection against			
	Analysis		Tampering	
	Static	Dyn.	Static	Dyn.
Encryption	F	N	F	N

- Problems:
  - Code in clear when executed
  - No dynamic verification (cfr. guards)

A New Scheme

- Scheme 1: *callee* = $D_{caller}(E_{caller}(callee))$ before *call*
- Scheme 2: Scheme 1 + re-encrypt after *return*
- Scheme 3: Scheme 2 + $E_{callee}(caller))$ after *call*, *caller* = $D_{callee}(E_{callee}(caller))$ before *return*

A New Scheme



Scheme Properties

- Code encryption

  → *confidentiality*

- Code dependencies (code as key) == implicit dynamic checking

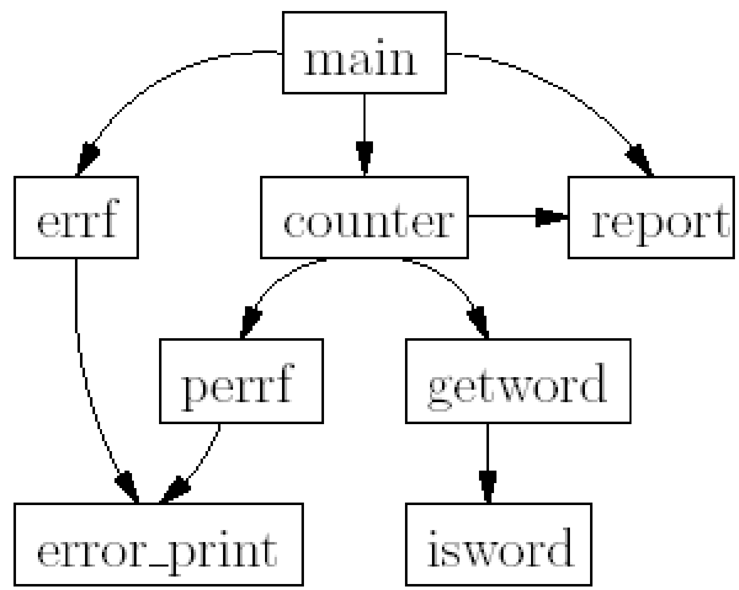
  → *data authenticity* (or *integrity*)

- Scheme

  → *Fault propagation* with nesting

Scheme Problems

- Multiple callers – which code as key ?
  - *n* callers
  - 1 out of *n*
  - …
- Or rely on E(*code*) as key
  - *n* callers
  - …



Scheme Cost

- Cost in speed → $C_s(P,P') = \frac{T(P')}{T(P)}$

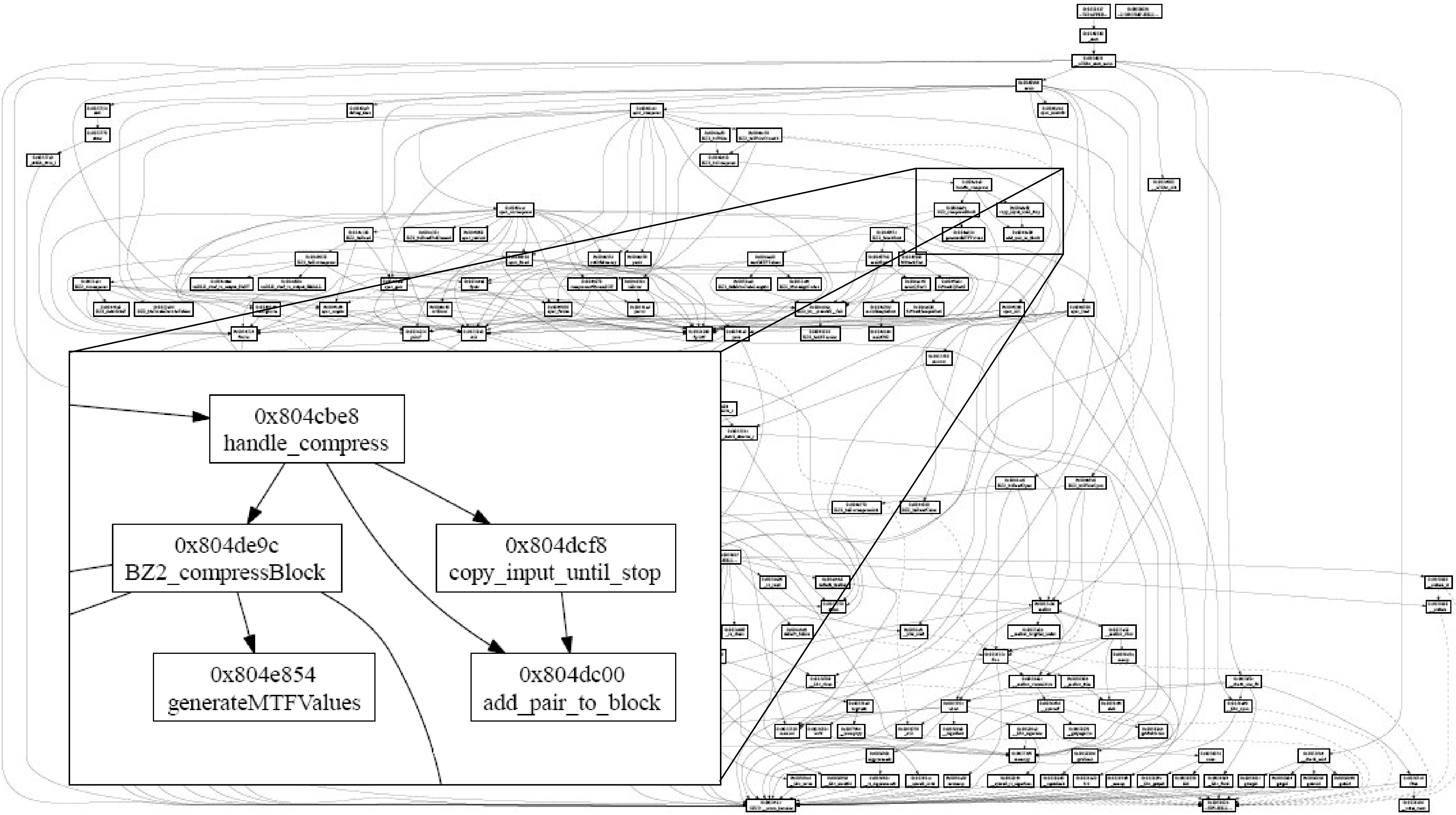
Program	Scheme 1	Scheme 2	Scheme 3
du	0.899	3.612	8.364
tar	0.822	1.339	2.783
wc	0.989	39.017	91.093

- After inlining the guards, C_s(wc) ~ 1000

Improvements

- *callee* = $D_{dominator}(E_{dominator}(callee))$
- Test framework
  - Diablo
  - SPEC CPU2006
- First results for Scheme 1
  - Bzip2 → 60 times slower

Dominators in a Call Graph



Further Improvements

- Avoid *hot code* (frequently executed)
- More optimal E() and D() functions
  – Size/speed versus security

- Obfuscation to hide *crypto guards*
- Interweave guard code with program code
- …

Conclusions

- Theory
  - Perfect security?
  - *"Attack on checksumming-based software"* by Wurster *et al.* IEEE-SSP'05
  - *"Strengthening self-checksumming via self-modifying"* by Giffin *et al.* ACSAC'05
  - …

Conclusions

- Practice
  - Another layer of security
    - Self-modifying code is hard to analyze
  - Security-versus-cost trade-off
    - Performance overhead