Recovery of high-level intermediate representations of algorithms from binary code

Alexander Borisovich Bugerya[1], **Ivan Ivanovich Kulagin**[2], Vartan Andronikovich Padaryan[2, 3], Mikhail Aleksandrovich Solovev[2, 3], Andrei Yur'evich Tikhonov[2]

[1] Keldysh Institute of Applied Mathematics of the Russian Academy of Sciences, Moscow, Russia
[2] Ivannikov Institute for System Programming of the Russian Academy of Sciences, Moscow, Russia
[3] Lomonosov Moscow State University, Moscow, Russia

Email: shurabug@yandex.ru, {i.kulagin, vartan, icee, fireboo}@ispras.ru

IVANNIKOV MEMORIAL WORKSHOP, VELIKIY NOVGOROD, SEPTEMBER 13-14, 2019

Flaws in application logic

- Flaws in application logic are hard to find
  - This requires developing a program behavior model before model violations can be detected
  - One of the approaches that do not require specifying a model is *dynamic taint analysis*
    - Its usage is hindered because false positives and negatives
    - The actual data transformations are typically not considered
  - To solve these problems, a human analyst is involved
  - Analyst actions are automated to a certain degree by various tools (Trawl, Ghidra, Binary Analysis Platform – BAP, etc.)

Flaws in application logic

- The order of analyst actions is based on expert knowledge, and often involves a *large amount of manual work*
- The hard degree and the result quality of manual analysis depend on *representation of the algorithm*
- Existed intermediate representations (IR) are unsuitable
  - Compilers IR
    (GENERIC, GIMPLE, RTL in GCC; LLVM IR; Program dependence graph)
  - IR of modeling machine instructions and binary analysis
    (Pivot/Pivot2[1], B2R2[2], REIL[3], MAIL[4], BAP (BIL)[5], BitBlaze[6], ESIL[7], etc.)

[1] – M.A. Solovev, M.G. Bakulin, M.S. Gorbachev, D.V. Manushin, V.A. Padaryan, S.S. Panasenko. **Next generation intermediate representations for binary code analysis.**
[2] – Jung, Minkyu and Kim, Soomin and Han, HyungSeok and Choi, Jaeseung and Kil Cha, Sang. **B2R2: Building an Efficient Front-End for Binary Analysis.**
[3] – T. Dullien and S. Porst. **REIL: A platform-independent intermediate representation of disassembled code for static code analysis.**
[4] – S. Alam, R. N. Horspool and I. Traore. **MAIL: Malware Analysis Intermediate Language: A Step Towards Automating and Optimizing Malware Detection.**
[5] – D. Brumley, I. Jager, T. Avgerinos and E. J. Schwartz. **BAP: A Binary Analysis Platform.**
[6] – D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam and P. Saxena. **BitBlaze: A New Approach to Computer Security via Binary Analysis.**
[7] – ESIL: Radare2 book. URL: https://radare.gitbooks.io/radare2book/content/disassembling/esil.html.

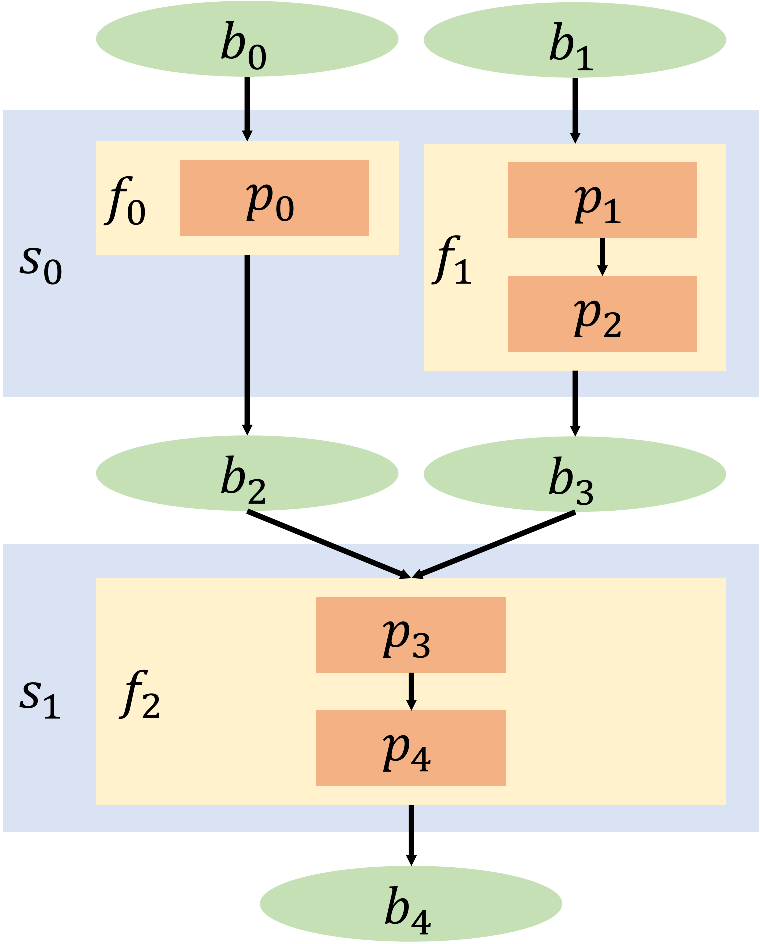
High-level algorithm representation

- Currently there is a lack of tools that could build from binary code *hierarchical flowchart-based algorithm representation* that is suitable for manual analysis
- It is needed to propose:
  - A high-level hierarchical representation of an algorithm based on flowcharts
  - Algorithm of whole-system binary code analysis that builds such a representation
- The proposed solution should not rely on any kind of code markup
- The proposed representation should be suitable for manual analysis and for implementing automatic data flow analysis algorithm in context of finding undocumented software features

HIERARCHICAL HIGH-LEVEL ALGORITHM REPRESENTATION

ISP RAS

- High-level hierarchical flowchart-based representation of an algorithm is based on hypergraph
- Representation has two kinds of nodes
  1. ***Points*** ($\boldsymbol{p_i}$) – represent an instruction executed at a certain trace step
  2. ***Buffers*** ($\boldsymbol{b_i}$) – represent a region of an abstract memory model (which can be an actual contiguous memory address range, a register or a part thereof) at a certain trace step
- Edges describe data dependencies
- ***Point*** nodes can be grouped into subsets – ***fragments*** ($\boldsymbol{f_i}$)
- ***Fragment*** nodes can be grouped into ***superblocks*** ($\boldsymbol{s_i}$)



Hierarchical High-Level Algorithm Representation

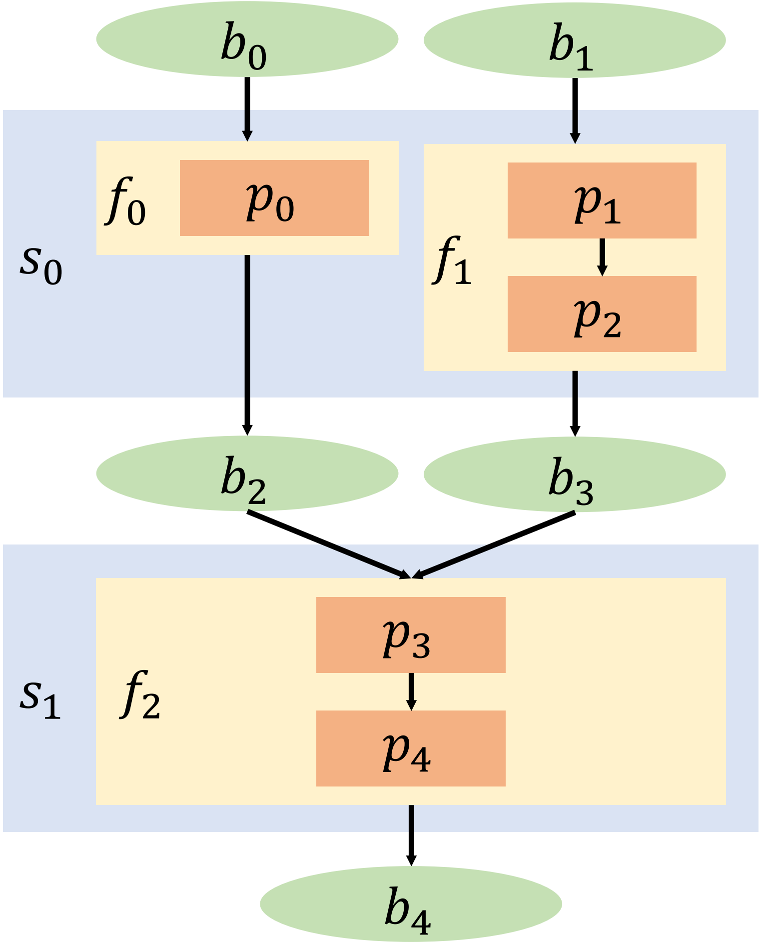
- Logically connected ***buffer*** nodes can be grouped into subsets called ***superbuffers***

  3. ***Fragment nodes*** $(\boldsymbol{f_i})$ – correspond to code fragments in the trace (linear step sequences such that there are no call or return instructions within them)

  4. ***Superblock nodes*** $(\boldsymbol{s_i})$ – correspond to instances of function calls and therefore can only contain fragments that belong to a single function instance

  5. ***Superbuffer nodes*** $(\boldsymbol{B_i})$ – logically connected buffer nodes

- Superbuffers and buffers correspond to data structures in the program and define interoperation interfaces between fragments and superblocks



CONSTRUCTION OF HIGH-LEVEL REPRESENTATION

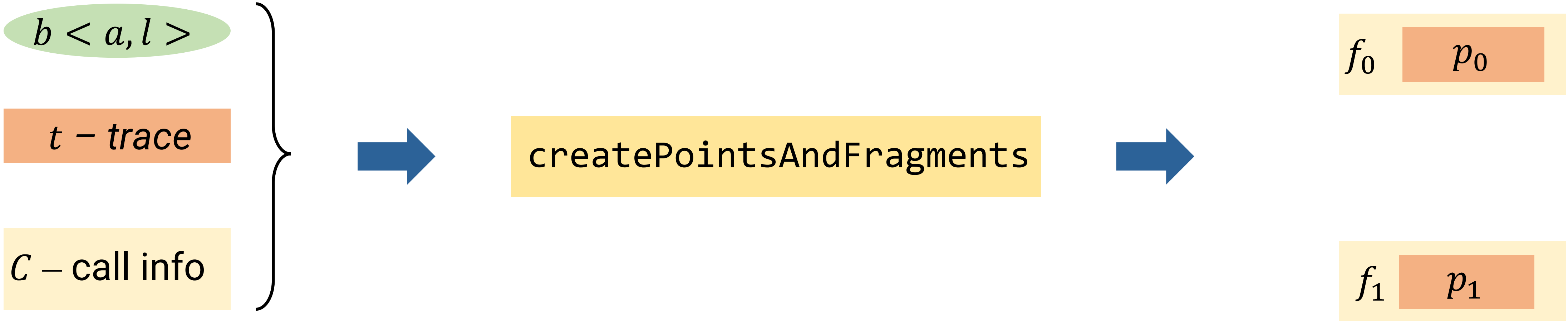
- The basis of constructing the high-level representation is the backward slicing algorithm used to track data flow in reverse step order
- Representation of an algorithm is built only from points (trace steps) that contribute to forming the result buffer
- Input of the construction algorithm:
  - ***Start buffer*** $b\colon < a, l >$ is a result buffer of the algorithm being analyzed (a – begin address of buffer; l – length of buffer)
  - ***Trace*** t where execution of the analyzed algorithm had been recorded
  - Functions call information C

CONSTRUCTION OF HIGH-LEVEL REPRESENTATION

- The construction algorithm performs two main steps:
  1) discovery of points in trace that belong to the algorithm forming the start buffer and their grouping into fragments (`createPointsAndFragments`)
  2) grouping fragments into superblocks (`createSuperblocks`)

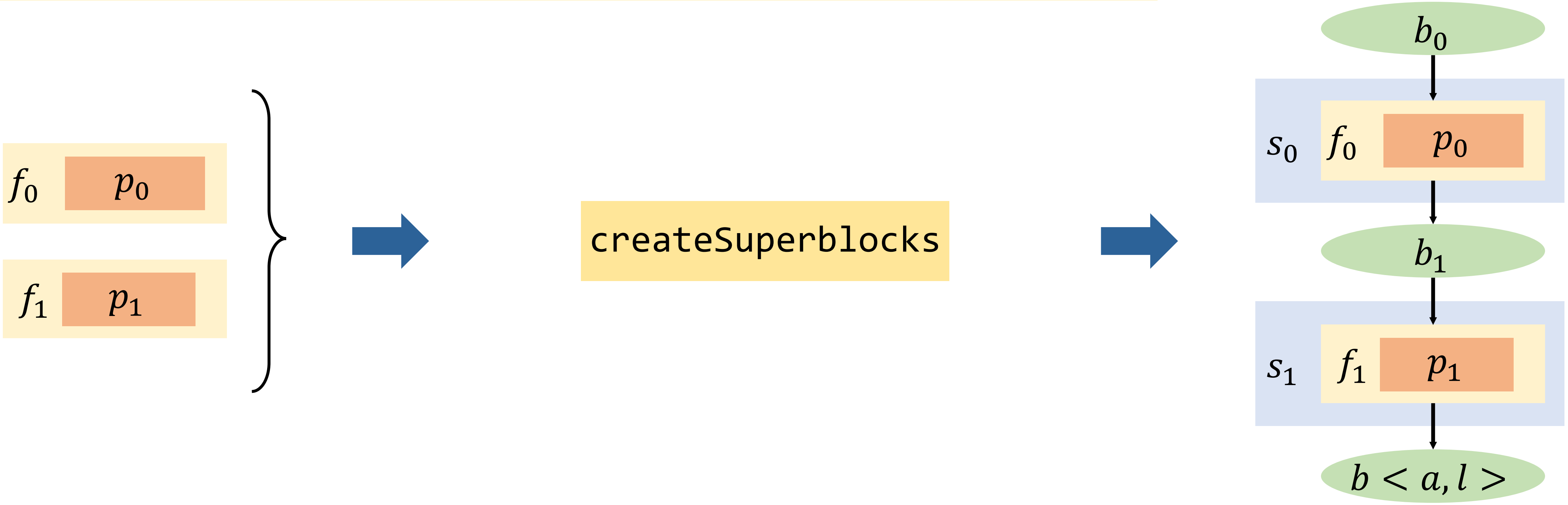
Construction of high-level representation

- The construction algorithm performs two main steps:
  1) discovery of points in trace that belong to the algorithm forming the start buffer and their grouping into fragments (`createPointsAndFragments`)
  2) grouping fragments into superblocks (`createSuperblocks`)



CONSTRUCTION OF HIGH-LEVEL REPRESENTATION

- The construction algorithm performs two main steps:
  1) discovery of points in trace that belong to the algorithm forming the start buffer and their grouping into fragments (`createPointsAndFragments`)
  2) grouping fragments into superblocks (`createSuperblocks`)

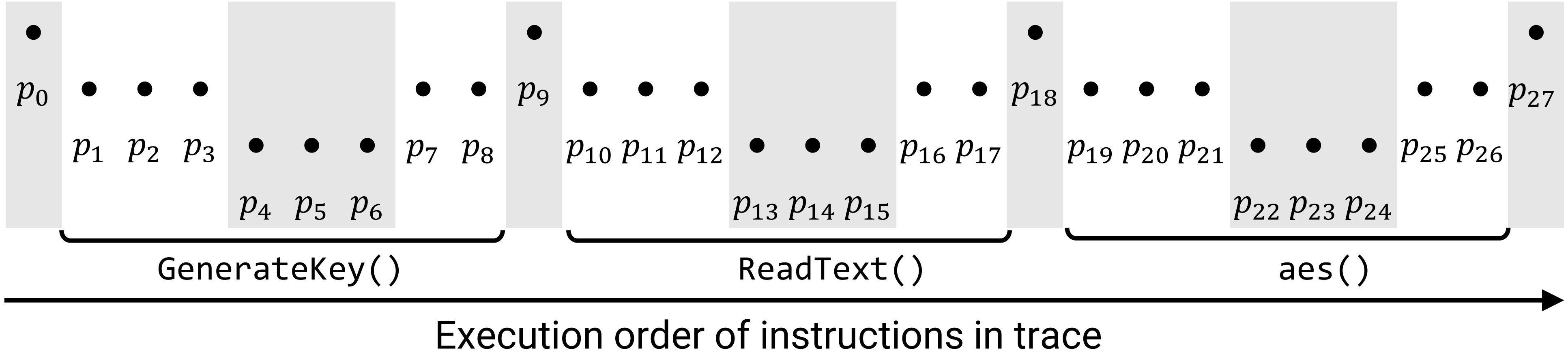


Example of construction of high-level representation

AES encryption program

```
key = GenerateKey()
text = ReadText()
cipher = aes(key, text)
```

Execution trace



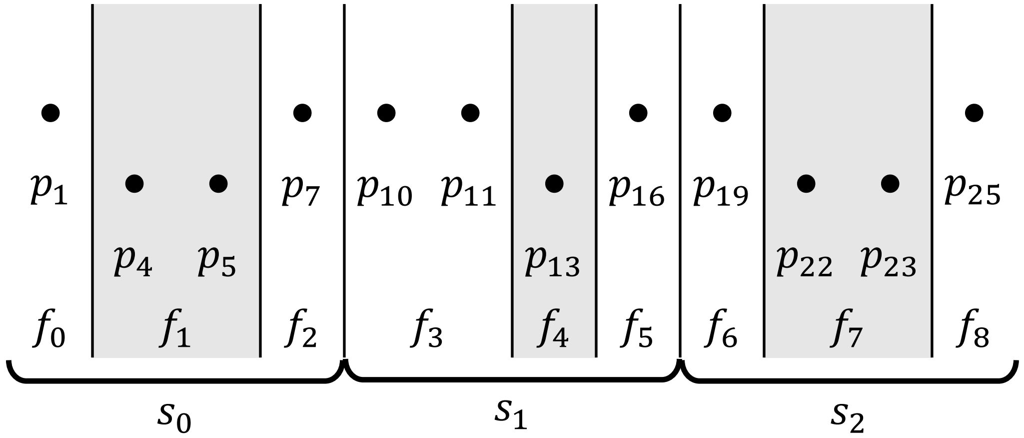
p_0 – `call GenerateKey()`

p_9 – `call ReadText()`

p_{18} – `call aes()`

Example of construction of high-level representation

- The point set corresponds to the backward trace slice for **cipher** buffer



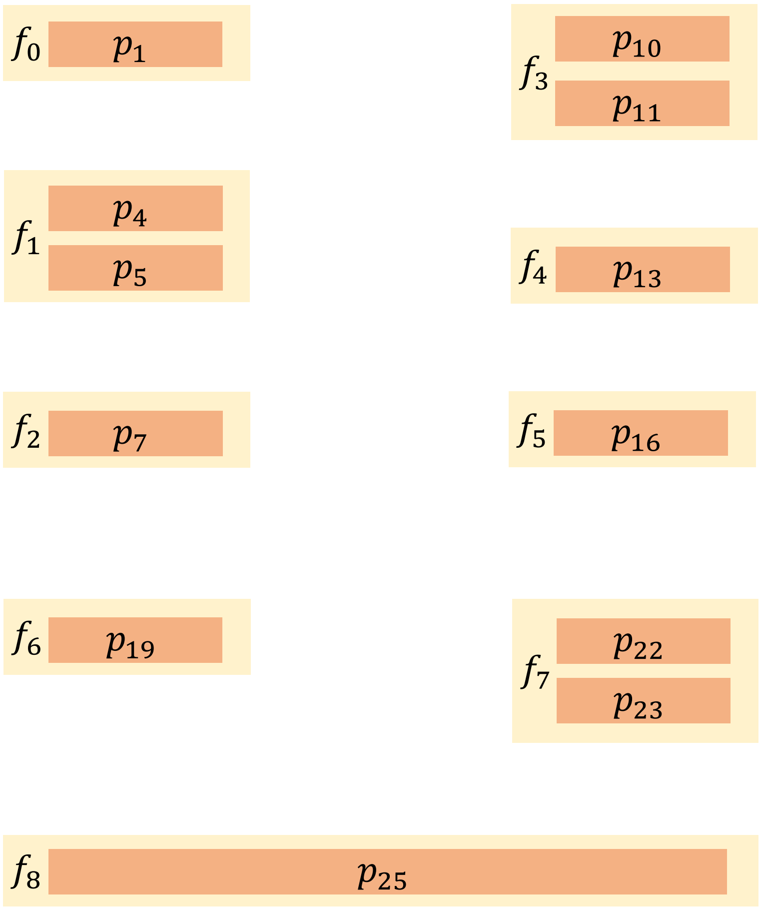
Example of construction of high-level representation

1. Grouping into fragments (`createPointsAndFragments`)

Example of construction of high-level representation

1. Grouping into fragments (`createPointsAndFragments`)

$f_0 = \{p_1\}, f_1 = \{p_4, p_5\}, f_2 = \{p_7\}, f_3 = \{p_{10}, p_{11}\}, f_4 = \{p_{13}\},$
$f_5 = \{p_{16}\}, f_6 = \{p_{19}\}, f_7 = \{p_{22}, p_{23}\}, f_8 = \{p_{25}\}$

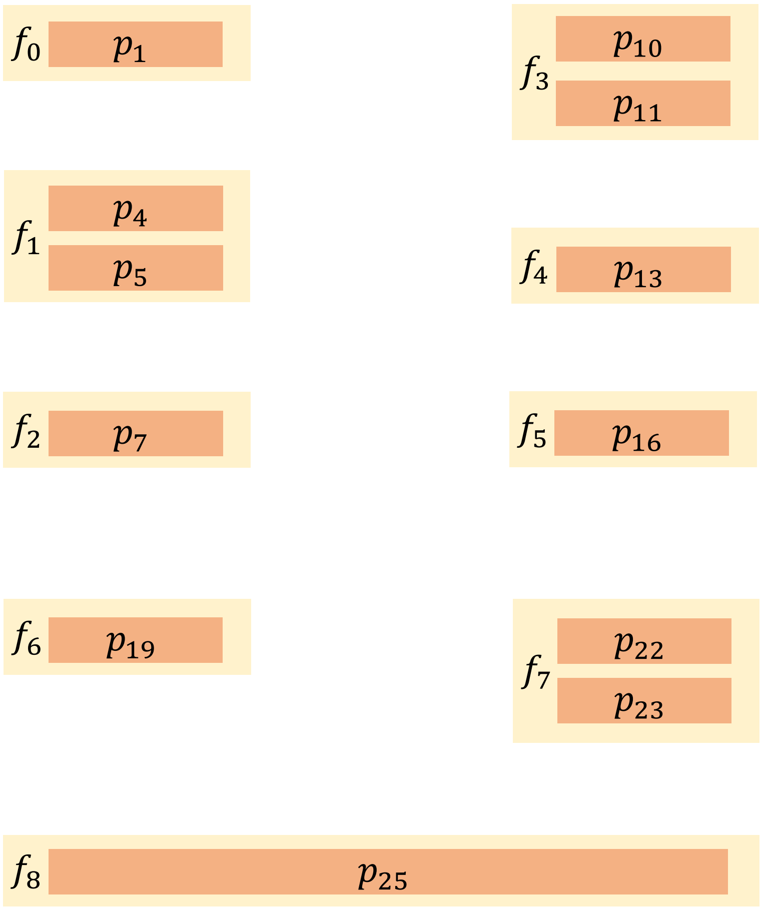


EXAMPLE OF CONSTRUCTION OF HIGH-LEVEL REPRESENTATION

1. Grouping into fragments (`createPointsAndFragments`)

$f_0 = \{p_1\}, f_1 = \{p_4, p_5\}, f_2 = \{p_7\}, f_3 = \{p_{10}, p_{11}\}, f_4 = \{p_{13}\},$
$f_5 = \{p_{16}\}, f_6 = \{p_{19}\}, f_7 = \{p_{22}, p_{23}\}, f_8 = \{p_{25}\}$

2. Grouping fragments into superblocks (`createSuperblocks`)



EXAMPLE OF CONSTRUCTION OF HIGH-LEVEL REPRESENTATION

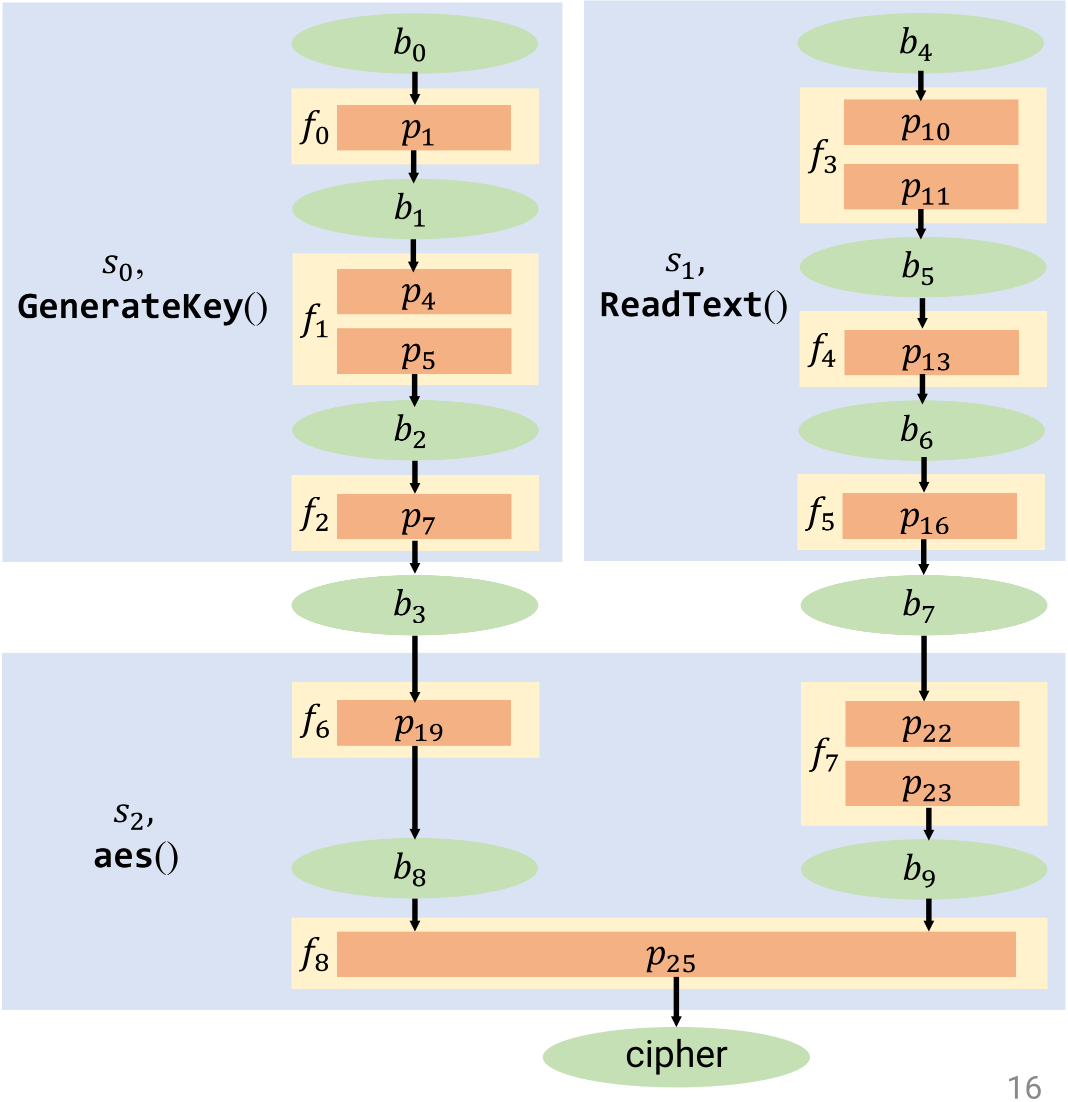
ISP RAS

1. Grouping into fragments (`createPointsAndFragments`)

$f_0 = \{p_1\}, f_1 = \{p_4, p_5\}, f_2 = \{p_7\}, f_3 = \{p_{10}, p_{11}\}, f_4 = \{p_{13}\},$
$f_5 = \{p_{16}\}, f_6 = \{p_{19}\}, f_7 = \{p_{22}, p_{23}\}, f_8 = \{p_{25}\}$

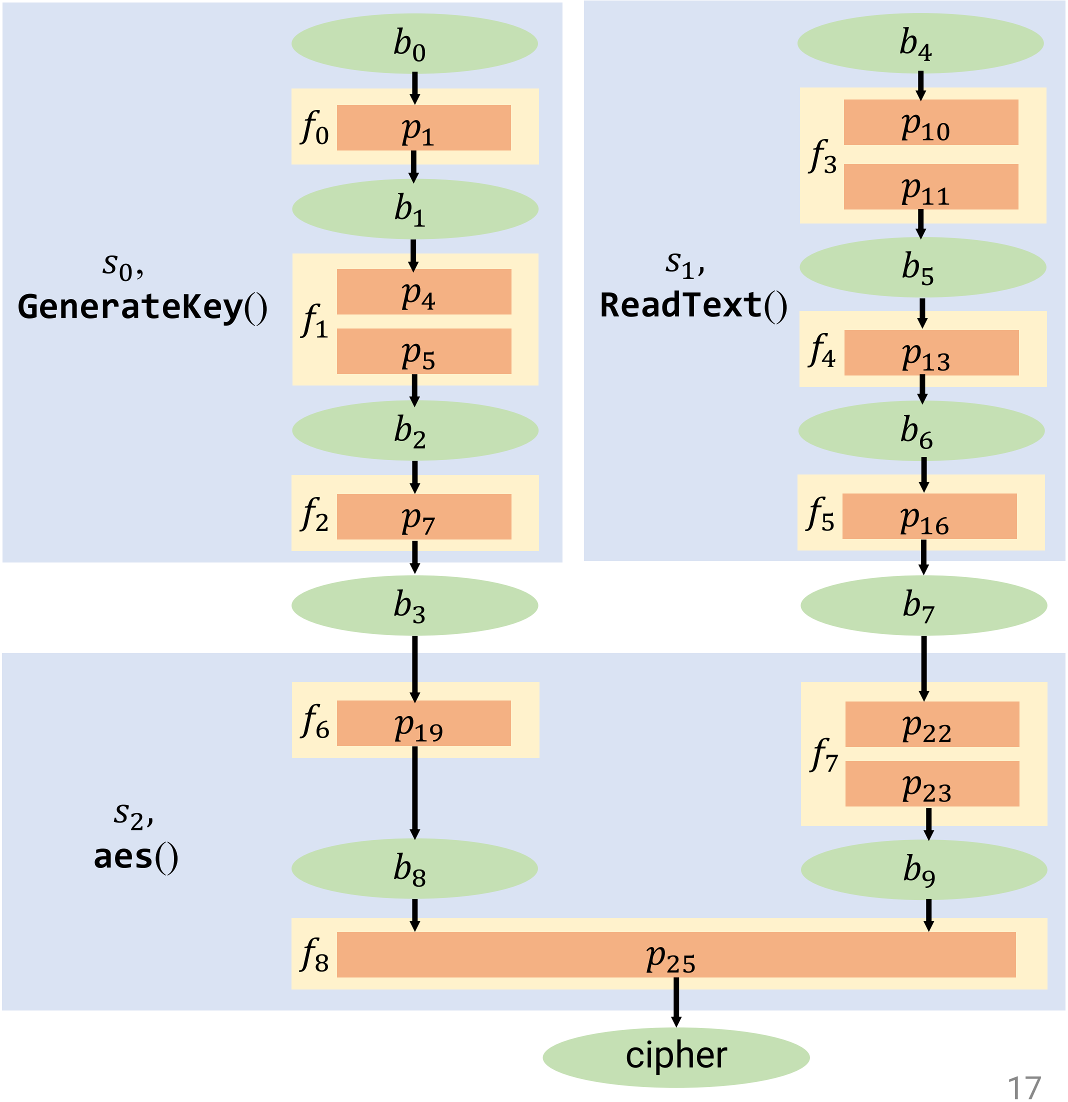
2. Grouping fragments into superblocks (`createSuperblocks`)

$s_0 = \{f_0, f_1, f_2\}, s_1 = \{f_3, f_4, f_5\}, s_2 = \{f_6, f_7, f_8\}.$



Example of construction of high-level representation

- The hierarchical organization of the representation is suitable for the algorithm research in manual mode (because excess details can be elided through folding fragments and/or superblocks)

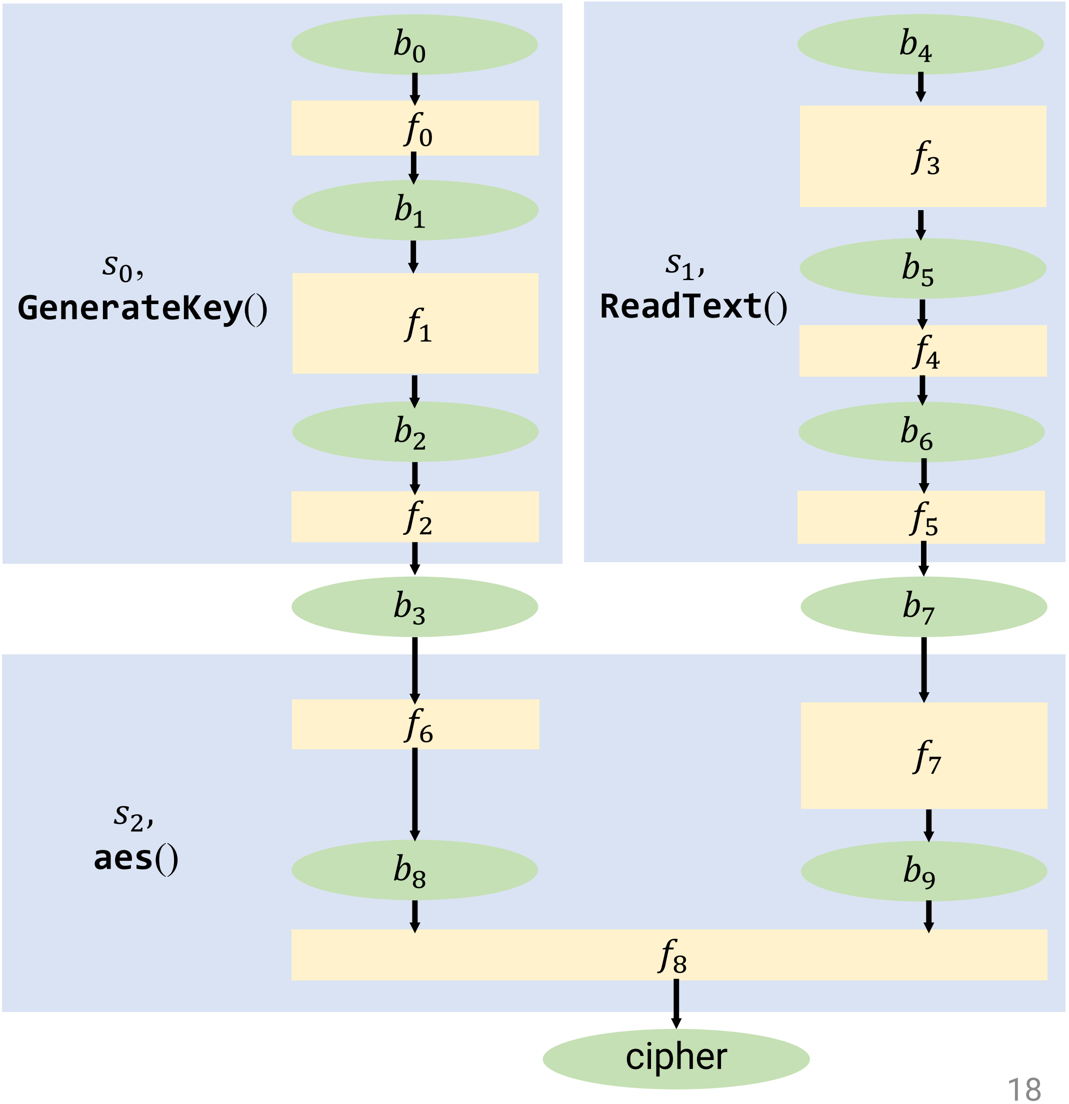


EXAMPLE OF CONSTRUCTION OF HIGH-LEVEL REPRESENTATION

ISP RAS

- The hierarchical organization of the representation is suitable for the algorithm research in manual mode (because excess details can be elided through folding fragments and/or superblocks)

Fold fragments $f_0,\ f_1,\ f_2,\ f_3,\ f_4,\ f_5,\ f_6,\ f_7,\ f_8$

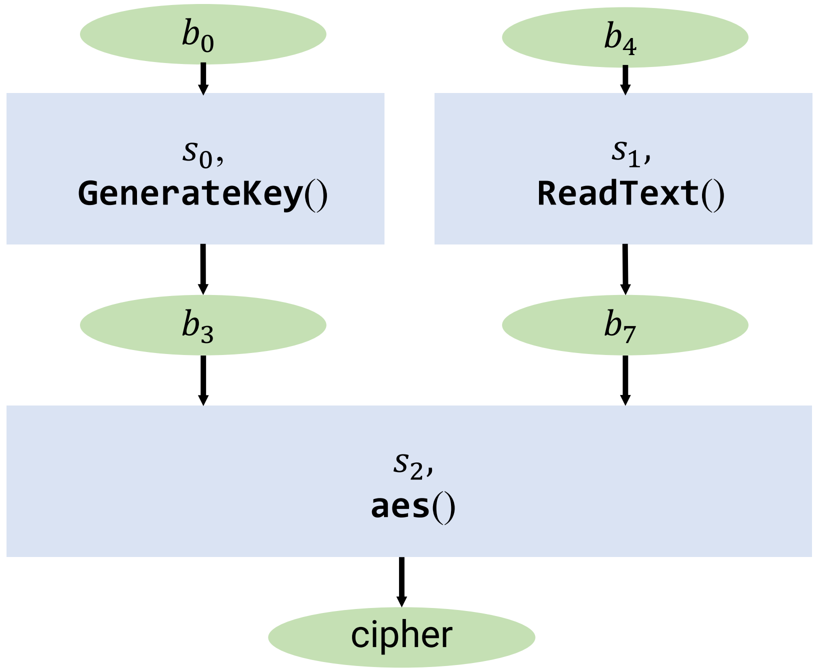


Example of construction of high-level representation

- The hierarchical organization of the representation is suitable for the algorithm research in manual mode (because excess details can be elided through folding fragments and/or superblocks)

Fold fragments $f_0,\ f_1,\ f_2,\ f_3,\ f_4,\ f_5,\ f_6,\ f_7,\ f_8$

Fold superblocks $s_0,\ s_1,\ s_2$



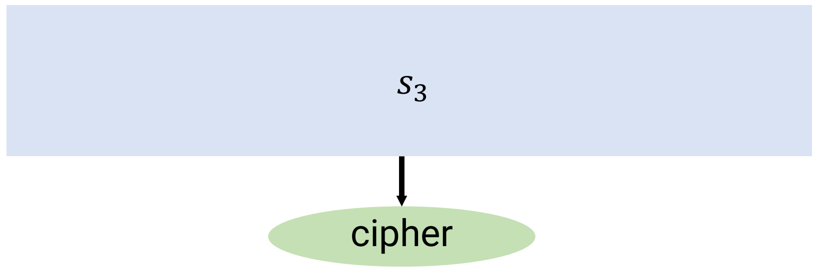
Example of construction of high-level representation

- The hierarchical organization of the representation is suitable for the algorithm research in manual mode (because excess details can be elided through folding fragments and/or superblocks)

Fold fragments $f_0,\ f_1,\ f_2,\ f_3,\ f_4,\ f_5,\ f_6,\ f_7,\ f_8$

Fold superblocks $s_0,\ s_1,\ s_2$

Fold entire diagram $\{s_0,\ s_1,\ s_2\} \rightarrow s_3$



CONCLUSIONS AND FUTURE WORKS

- The hierarchical high-level representation of a program’s algorithm has been proposed
- The representation is based on a hypergraph and permits analysis in manual and automatic settings
- Algorithm of whole-system binary code analysis that builds such a representation has been proposed

- Future works:
  - Improving the quality of the representation by identifying high-level language constructs (such as conditional and loop statements, etc.) and recovering structural and type information for program variables
  - Development of automatic methods of analysis of an algorithm’s properties based on its high-level representation

Thank you for your attention!

Alexander Borisovich Bugerya[1], **Ivan Ivanovich Kulagin**[2], Vartan Andronikovich Padaryan[2, 3], Mikhail Aleksandrovich Solovev[2, 3], Andrei Yur'evich Tikhonov[2]

[1] Keldysh Institute of Applied Mathematics of the Russian Academy of Sciences, Moscow, Russia
[2] Ivannikov Institute for System Programming of the Russian Academy of Sciences, Moscow, Russia
[3] Lomonosov Moscow State University, Moscow, Russia

Email: shurabug@yandex.ru, {i.kulagin, vartan, icee, fireboo}@ispras.ru