K-Hunt++: Improved Dynamic Cryptographic Key Extraction

Thomas Faingnaert
thomas.faingnaert@ugent.be
Ghent University
Ghent, Belgium

Willem Van Iseghem
willem.vaniseghem@ugent.be
Ghent University
Ghent, Belgium

Bjorn De Sutter
bjorn.desutter@ugent.be
Ghent University
Ghent, Belgium

Abstract

We identified several weaknesses in the state-of-the-art cryptographic key extraction algorithm, K-Hunt. It cannot handle code in which key loading and use are spread apart, has problems with modes such as AES CBC that use small data buffers of constant size, and with complex apps in which functionality handles both the key and data. K-Hunt++ overcomes those weaknesses. We demonstrate it on two apps that trigger them and present an ablation study and qualitative analysis of its robustness in the face of obfuscation.

CCS Concepts

• **Security and privacy → Software reverse engineering**; **Cryptanalysis and other attacks**.

Keywords

Dynamic binary code analysis; cryptographic key identification

ACM Reference Format:
Thomas Faingnaert, Willem Van Iseghem, and Bjorn De Sutter. 2024. K-Hunt++: Improved Dynamic Cryptographic Key Extraction. In *Proceedings of the 2024 Workshop on Research on offensive and defensive techniques in the context of Man At The End (MATE) attacks (CheckMATE '24), October 14–18, 2024, Salt Lake City, UT, USA.*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3689934.3690818

1 Introduction

Cryptographic keys play a crucial role in ensuring privacy and confidentiality. The use of hardcoded private keys is considered a bad practice that introduces significant security risks [9, 17]. Best practices recommend the use of secure key management solutions, where keys can be dynamically generated and securely stored, rather than embedding them in code. Dynamic key usage can also be insecure, however, when the keys are generated deterministically, insecurely negotiated, or recoverable in other ways [19].

Quite some research has been invested in automated analyses to identify cryptographic primitives in apps, and to extract static and dynamic keys. These techniques are not only used for security analysis, but also in man-at-the-end attacks that aim for reverse engineering software or tampering with it to make unauthorized use of it. Software obfuscations aim to mitigate such attacks [24].

Static analyses to localize crypto primitives and keys have been based on pattern matching of data flow [18, 22] and control flow [5]. Library detection tools such as F.L.I.R.T. [13] can also be used to detect crypto library functionality linked statically into an app.

Dynamic analyses have also built on various forms of data flow analysis and tracking [7, 11, 31]. Instruction mix properties and loop properties have been used [14], as well as a-priori knowledge about implementations and features of cryptographic functionality, such as the used data structures [29], and the avalanche effect [20].

K-Hunt [19] is a system for identifying insecure key usage in binary executables. It includes a state-of-the-art dynamic key extraction technique. As it does not rely on signatures to identify crypto operations, it can target unknown and proprietary crypto algorithms. K-Hunt was demonstrated to locate the keys in symmetric ciphers, asymmetric ciphers, stream ciphers, and digital signatures, both in standardized algorithms and in proprietary ones. Based on that localization, K-Hunt discovered insecure keys in 22 out of 25 evaluated programs including several well-known crypto libraries.

No complete implementation of K-Hunt is available however: while the K-Hunt paper does reference a GitHub repository, that repository does not contain the fully implemented version of the tool described in the paper. The authors confirmed to us that this was intentional, as K-Hunt had since been acquired by a company, which precluded them from publishing the full source code.

When we tried to reimplement K-Hunt as part of our research into the modeling of reverse engineering strategies, we stumbled on some shortcomings of the approach: namely that it cannot handle code in which key loading and key use are spread apart, that some of its heuristics fail for certain encryption/decryption modes such as AES CBC, in which the result of each encryption operation is a small buffer of constant size, and that it can fail on complex applications in which functionality is reused in both the key and the data handling, with the complementary heuristics not being robust enough to overcome cheap countermeasures.

Section 2 presents and evaluates K-Hunt++, an extension of K-Hunt that overcomes its shortcomings, on two programs. Section 3 presents an ablation study of K-Hunt++'s heuristics, before a discussion on the robustness of K-Hunt++ in the face of software obfuscations in Section 4. Section 5 draws conclusions.

2 K-Hunt++

2.1 Design

K-Hunt++ extracts cryptographic keys from applications in four stages as shown in Figure 1. Each stage aims to identify a milepost: three intermediate mileposts, and a final milepost that consists of the ultimate target, namely the cryptographic keys being used in the program. All techniques and heuristics in K-Hunt++ that were inspired by K-Hunt are indicated in an italic font in the figure. Like

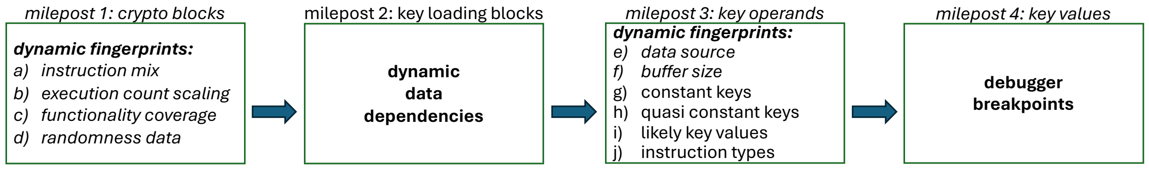

Figure 1: Four stage K-Hunt++ pipeline to identify four mileposts. Concepts that originate from K-Hunt are in italic [19].

K-Hunt, K-Hunt++ is a fully automated, dynamic approach. Its first three pipeline stages analyze multiple executions on different inputs. For extracting the dynamic information, instrumentation tools such as Intel's Pin [15, 21] can be used, as well as simulators such as QEMU [4]. We used Pin. The fourth stage relies on a debugger script or instrumentation to extract keys from an actual program execution. All stages are implemented in Python notebooks.

2.1.1 Milepost 1: crypto basic blocks. In stage one, K-Hunt++ uses a set of heuristics to identify the set of basic blocks that likely perform crypto operations. These heuristics come down to fingerprints that, when combined, discriminate basic blocks performing crypto operations from other blocks. The four fingerprints are:

(a) *Caballero heuristic*: crypto basic blocks have a high ratio of arithmetic, bitwise logic, or AES instructions [6].
(b) *Linear scaling of execution count*: the number of executions of crypto basic blocks scales linearly with input size.
(c) *Functionality coverage*: if the target application allows for disabling the cryptography, or for changing the used cipher, we use coverage information in multiple runs with different ciphers enabled/disabled to prune the search space.
(d) *High randomness*: the data consumed or produced by the instructions in the crypto basic blocks have high entropy.

All of these heuristics were already included in K-Hunt [19]. They target fundamental properties of cryptographic primitives: when encryption, decryption, or signing primitives are invoked, XORs or similar operations are executed to handle an amount of data that scales with the size of the input plaintext or ciphertext. It are these computations that this stages identifies, as their operands are the keys and subkeys we aim to extract.

We implemented (a) with a Pintool that partitions the stream of executed instructions into basic blocks using Pin's heuristics. The tool computes the ratio of targeted to other instructions in each block, and then labels blocks as relevant or irrelevant based on a threshold value. For (b) and (c) a second Pintool collects execution counts of all executed basic blocks for multiple runs on varying input sizes and with the targeted cryptographic functionality enabled/disabled. For (d) yet another Pintool collects the sequence of values being loaded by each of the memory access instructions and computes the Shannon entropy of that set per instruction. This Pintool also tracks how many different memory locations are accessed by each instruction in the program, thus measuring the sizes of the buffers that are accessed by those instructions. This information will be used in a later stage of the pipeline.

For each identified basic block, we perform a linear regression analysis on the block's execution counts using r^2 as a metric of linearity of the execution count as a function of the program input length. Our metric also takes into account the slope of the line of best fit to filter out blocks which are executed a constant number of times. We experimented with two selection algorithms: the first excludes all basic blocks that fail on any of the criteria, using thresholds for the entropy and r^2. The second computes a weighted average of the entropy, r^2, and the two binary values of the Caballero heuristic and the coverage analysis, on the basis of which it sorts all blocks from most to least likely crypto. We used the latter for all experiments reported later in this paper.

2.1.2 Milepost 2: key loading blocks. To identify the keys used in the blocks of milepost 1, fingerprints will again be used, as will be discussed in the next section. Some fingerprints concern not the computational operations such as the XORs, but the memory buffers from which their operands are read. Those fingerprints are expensive to collect, i.e., the amount of data to collect and the processing thereof during the tracing introduces huge slowdowns. To minimize that slowdown, the required data should only be collected for the relevant buffer accesses, not for all memory accesses in the whole program. In other words, we want to limit further fingerprint collection through tracing to only the operations that read data that is actually used in the already identified basic blocks.

Those read operations do not necessarily need to be part of the basic blocks identified as crypto basic blocks in stage 1, however. For example, a key might be read into a register before a loop, and then used within the loop. The goal of this stage is to identify the basic blocks that potentially load the used keys.

We hence identify the dynamic data dependencies to the blocks identified in stage 1, by observing which operations those blocks depend on during the tracing, i.e., of which they consume data. The set of blocks from stage 1 is then expanded with the blocks at the sources of these data dependencies. For this, we again rely on a Pintool. This expansion was missing from the original K-Hunt.

2.1.3 Milepost 3: key operands. Within the blocks identified so far, stage 3 aims to discriminate the instructions reading (sub)keys from those reading input data. At the same time, this stage identifies which operands of those instructions hold the (sub)keys.

The stage relies on a set of complementary heuristics to identify the instructions reading from memory buffers and to differentiate between instructions likely loading plaintext or ciphertext data on the one hand and those loading keys on the other hand:

(e) *Data source*: a key is typically initialized by a key derivation function or from a random number generator, whereas the ciphertext or plaintext input typically originates from a specific file or network socket.
(f) *Buffer size*: key buffers are typically narrower than data input buffers. Moreover, key buffer sizes are typically constant, independent of the input data. Here, buffer *size* refers to the number of unique addresses that an instruction accesses.

This heuristic is not always applicable, however. In AES CBC encryption, e.g., the ith encryption input plaintext$_i$ $\oplus$ ciphertext$_{i-1}$ is stored in a small buffer of constant size.

(g) *Constant keys*: Hardcoded keys do not vary over different runs, while chosen plaintexts or ciphertexts do. When varying passwords are used, however, derived keys will vary.

(h) *Quasi constant keys*: keys do not vary within one run, unlike the blocks of text being encrypted or decrypted.

(i) *Likely keys values*: keys are likely not small constants that have (many) 0-bytes at the most significant position, valid memory addresses, or standardized constants (e.g., S-Boxes).

(j) *Instruction types*: key buffers are accessed less likely by control flow instructions, compares, and pops.

Heuristics (e) and (f) are reused from K-Hunt [19], the remaining ones are novel: no such fingerprints or heuristics are discussed or mentioned in the K-Hunt paper [19].

For (e) we implemented two approaches. In a first approach, we rely on a taint tracking tool that we derived from the work by Yadegari et al. [32, 33]. Specifically, we extended the original taint tracker implementation such that it can be configured to taint data that enters the program through specific system calls, such as `open` that is used to open input data files, and `read` that is then used to read their data. Importantly, Yadegari's offline taint tracking tool, as well as our adaptation, operates on detailed trace files that include a list of all executed instructions as well as all of their operands, which can easily be produced with a Pin tool.

Producing such files is not feasible for long running processes, however, as the files grow too big and their processing becomes impractically slow. While all of the mentioned fingerprints and heuristics can be obtained on relatively small inputs, and hence short running cryptographic operations, the code preceding the cryptographic operations might run much longer. An important case for us is key stretching in key derivation functions such as Password-Based Key Derivation Function 2 (PBKDF2) [16]. Key stretching is the iterative, recursive hashing of a password to derive a key in a time-consuming way, i.e., with very large iteration counts. It is deployed to slow down brute-force attacks and dictionary-based attacks, e.g., on encrypted disks or other information stores protected (and encrypted) with passwords. If key stretching cannot be disabled or shortened through a configuration option, this can impede the collection of full, detailed traces.

The authors of K-Hunt also observed that fine-grained taint tracking incurs high performance overheads. They hence provided an alternative, in the form of coarse-grained, function-level dynamic taint tracking [19]. In essence, their analysis labels functions as tainted by local inputs (from opened files), by remote inputs (obtained through sockets), or both or none of those two. During the execution of the program, functions get the taint labels if they consume data from those taint sources, or if they consume data produced by tainted functions. While this function-level taint analysis proved to work on the applications on which K-Hunt was originally evaluated, we think it is rather trivial to break this analysis. It suffices to insert invocations of memcpy()-like functions on buffers propagating keys and on buffers propagating input data at the appropriate points to make the analysis consider both buffers, and hence the operations loading keys, as tainted by the data source.

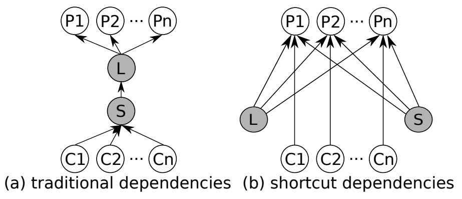

Figure 2: The data dependencies in an example program. Shaded nodes represent memory operations in `memcpy()`.

We hence developed an alternative approach that is not as vulnerable, and that is still efficient because it relies solely on dynamic data dependencies, which already need to be tracked for milepost 2 anyway. In our approach, we build a data dependency graph (DDG) that incorporates all dependencies observed in a tracing run. In this DDG, we then simply compute the distances from the system call through which the input data enters the program to the operations that load key or input data from key or data buffers in the crypto and key loading blocks. The operations with the shortest distance to the system calls are assumed to be loading from data buffers, the others (which typically have distance infinity, as there is no path connecting the system calls and those loads in the DDG) are then considered as loading from key buffers.

This approach can only work, however, if the data dependencies are tracked accurately. In practice, many data-copying operations, such as those in `memcpy()`, have high fan-in or fan-out in dynamic DDGs. In a DDG that incorporates all direct dependencies that occurred in a full execution, this can obscure the relevant relations between computational producers (or system calls producing data) on the one hand and their consumers on the other hand. For example, Figure 2(a) shows a conventional DDG of a program containing n producer-consumer pairs. In this program, all data propagates from producers to their consumers via a single load and a single store operation in (a simplified) `memcpy()` that is invoked at n call sites. When those load and store operations are handled like any other operation, the resulting DDG of Figure 2(a) does not reveal that each `Ci` only depends on its corresponding `Pi`. Instead, it mixes all of them up. This would make our approach fail, just like the insertion of memcpy() operations would make the function-level taint tracking of K-Hunt fail.

In our Pintool that collects data dependencies, we solve this by tracking so-called shortcut dependencies: for each value stored in any location in the program state, our tool also tracks the last instruction that actually produced that value with arithmetic, logic, or any other kind of computations, or in a system call. The dependencies of the resulting DDG of Figure 2(b) are then added to the DDG on top of the conventional dependencies, and distances are computed using the shortcut dependencies.

Finally, for heuristics (f–i) we again rely on simple Pintools, and (j) is implemented directly in our Python notebook. Features (e) to (j) are then used in a priority function that ranks the instructions and their operands in the identified basic blocks.

2.1.4 *Milepost 4: key values.*

Finally, we use the LLDB [30] debugger's scripting capabilities to insert breakpoints on the instructions that got prioritized on top in the previous stage, and to print the values of the relevant memory operands of those instructions in an execution of the program. When an encryption algorithm uses subkeys derived from the main key, these breakpoints will be triggered in the order in which the subkeys are used in the program. The subkeys will hence be reported in the order in which they are used, which eases the reconstruction of the main key.

Alternatively, the subkeys can be obtained with tracing. The Pintools collecting values for heuristics (g–i) then need to adapted to record also in which order different values are loaded.

2.2 Evaluation

We evaluated K-Hunt++ on two benchmark applications: 7-Zip [25], a file archiver which also supports file encryption, and the GNU Privacy Guard (GPG) [27], a libre implementation of the OpenPGP standard for private and authenticated data communication.

As stage 3 ranks instructions according to their likelihood of loading keys, we report the rank of the relevant instructions.

2.2.1 7-Zip. We compiled p7zip 16.02 in Linux, with the default compilation options from the project's `makefile.linux_amd64`. We did not use `makefile.linux_amd64_asm` to avoid Intel's AES-NI instructions, which would make localization too easy.

7-Zip uses 2^{19} iterations of SHA256 key stretching to derive the cryptographic key from the user-provided password, which prevents the use of Yadegari's fine-grained taint-analysis for heuristic (e). Moreover, 7-Zip uses AES256 encryption in a CBC scheme, in which intermediate data are stored in a small buffer whose size does not vary for different input sizes. This impacts heuristic (f).

In the inner loop of the AES encryption routine, the roundkeys are read cooperatively by a set of 16 instructions:

(1) Four instructions, each reading 4 bytes of the initial roundkey
(2) Four instructions, each reading 4 bytes of the even-numbered roundkeys (round 2, 4, 6, 8, 10, 12, and 14) in each execution
(3) Four instructions, each reading 4 bytes of the odd-numbered roundkeys (round 3, 5, 7, 9, 11, 13) in each execution
(4) Four instructions, each reading 4 bytes of the final roundkey

As not all these instructions are executed the same number of times, the part of the priority computed in the third pipeline stage that takes into account the values read by instructions (i.e., (g)–(i)), differs. To ensure this does not lead to wildly different overall priorities, we calculate these components of the priority function separately and cap them at a threshold. This threshold is determined with K-means clustering with 2 clusters on the natural logarithm of the priority, and taking the midpoint between the 2 cluster centers.

The result is that these 16 relevant instructions are ranked with the same top priority by K-Hunt++. We manually verified that stage 4 correctly reported the AES roundkeys when breakpoints are set at the instructions with top priority.

2.2.2 GPG. We used the gpg binary of the latest LTS versions of GPG and its dependent libraries: GPG 2.2.29, libassuan 2.5.5, libksba 1.6.0, libgcrypt 1.8.8, libgpgerror 1.42, nPth 1.6, bzip2 1.0.8, and zlib 1.2.11. We used the default compilation options for all of them, with the exception of disabling AES-NI support for libgcrypt.

GPG uses a variable number of iterations of SHA1 key stretching in its key-derivation function string-to-key (S2K). With a command-line option, we fix the number of iterations to 1025. Also worth

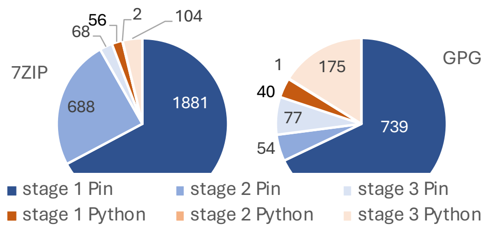

Figure 3: Execution times (seconds) pie charts for stages 1–3.

mentioning is that, by default, S2K uses salting, which results in a different encryption key in every program execution, thus voiding heuristic (g). Moreover, in the binary we observed that the key buffer reads happen in a different basic block than the actual cryptographic computation. Stage 2 of our pipeline correctly identifies that block.

We used the default AES256 in a CFB scheme, i.e., the *i*th input of the AES encryption routine is ciphertext$_{i-1}$. Additionally, for integrity verification of the encrypted message, GPG computes a Modification Detection Code (MDC), which is a SHA1-hash of the plaintext data. This has the interesting side effect that SHA1 is executed on both data inputs and key inputs. The code used to generate the keys is hence tainted by the data, which impacts heuristic (e). This implies that a context-sensitive, fine-grained taint-analysis needs to be used, or additional information such as the distance in the DDG as discussed in Section 2.1.3.

In this program, there is another reason for requiring either fine-grained taint information or DDG distance information. The GPG binary includes a number of self-checks that invoke `memcmp()` on input/output-related data as well as on key-related data. `memcmp()` performs computations on data rather than simply moving them around such as `memcpy()`, so the shortcut dependencies discussed in Section 2.1.3 do not provide the required context-sensitivity. Using distance in the DDG proved to be a sufficient alternative.

In the inner loop of the AES encryption routine, the roundkeys are read cooperatively into XMM registers by 3 instructions:

(1) One that reads the 16-byte initial roundkey
(2) One that reads the 16-byte roundkey for the 13 middle rounds, one in each execution
(3) One that reads the 16-byte roundkey for the final round

As for 7-Zip, these are not executed the same number of times, leading to differences in priority for the heuristics (g)–(i). We thus again apply the K-means clustering for these components of the priority function. K-Hunt++ then marks these 3 instructions with the same top priority, with all other instructions getting a strictly lower priority. We manually verified that stage 4 correctly reported the AES roundkeys when breakpoints are set for the 3 instructions.

2.2.3 Efficiency. Figure 3 shows the execution times the first 3 stages (stage 4 is negligible). All heuristics are used in this experiment. For heuristic (e), only our DDG-based analysis is used, as that sufficed, not the fine-grained taint-tracking derived from Yadegari et al. Unlike K-Hunt, of which the implementation is claimed to be completely online [19], our implementation is hybrid offline/online: first a Pinball is generated [15], which is an efficient record on which to replay the multiple Pintools that collect actual data. Our Pintools output the relevant data rather than traces, and that data is stored in a graph database (DB). The times in Figure 3 are split over instrumentation and tracing activities (Pinball generation and Pintool execution) shown in blue on the one hand and data analysis (Python code including queries on data stored in the graph DB) shown in orange on the other hand. All times were measured on an Intel(R) Core(TM) i7-10700 CPU running at 2.90GHz. The tracing activities clearly dominate the overall execution time.

While K-Hunt's authors claim their fully online approach is more efficient than other key extraction techniques, they also rely on Pin for heuristics (a)–(f). They do not report execution times, but on the basis of their use of Pin, and tracing times dominating K-Hunt++, we conjecture that K-Hunt's times will be of the same order of magnitude (tens of minutes) as K-Hunt++'s.

2.3 Comparison with K-Hunt

On GPG, K-Hunt would fail completely because it lacks KHunt++'s stage 2 that finds key-loading instructions outside the blocks that contain the crypto computations. Moreover, we have a strong suspicion that K-Hunt's course-grained taint analysis in support of heuristic (e) will also fail even if a stage 2 would have been added, due to the complexity of the GPG binary, in particular its use of multiple functions on both input-related data and key-related data. Not having the code of K-Hunt, we cannot validate this.

On 7-Zip, with its AES CBC mode, heuristic (f) fails. This illustrates that (e) and (f) can easily fail, even without obfuscations being deployed. As K-Hunt has no fallback heuristics such as (g)–(j), we have to conclude that K-Hunt is very brittle. This observation might at first sight contradict the results its authors reported on 10 libraries and 15 binaries [19]. Those binaries did not include GPG, however, and while their libraries do include libgcrypt on which GPG relies, K-Hunt was only evaluated on three binaries that each wrapped a single cryptographic primitive from libgcrypt, thus not featuring the complex interplay with SHA1 and self-tests of GPG. In short, K-Hunt was not tested on applications of the level of complexity of the main binary of GPG.

K-Hunt++, by including its stage 2 and by relying on more, better, and complementary heuristics in stage 3, is clearly less brittle.

3 Ablation Study

We studied all $2^4 - 1 = 15$ combinations of heuristics (a) – (d) for stage 1, and all $2^6 = 64$ combinations of heuristics (e) – (j) for stage 3. Detailed numeric results are listed in Appendix A. Here we focus on the main qualitative outcomes. For the ablation study of stage 1, we enabled all heuristics in stage 3, and vice versa.

3.1 Stage 1

On a relatively simple app such as 7-Zip, omitting one, two, or even three of the four heuristics (a)–(d) does not negatively impact the end results, i.e., the extracted keys. The combined stage 3 heuristics (e)–(j) are so good that, when they are all deployed, they still rank the 16 relevant instructions exclusively at the highest priority, thus enabling correct key extraction with perfect recall and precision. When multiple stage 1 heuristics are omitted, the number of basic blocks identified as potential mileposts 1 and 2 can grow rapidly, however, resulting in many more blocks being instrumented for stage 3, which significantly slows down that stage.

For a complex app such as GPG, the result is somewhat different. Most importantly, when any combination of up to three stage 1 heuristics are omitted, but all stage 3 ones are still deployed, the three key-loading instructions still get ranked with the top priority by stage 3. Depending on which stage 1 heuristics are omitted, additional instructions also get ranked at that top priority. In other words, omitting up to three stage 1 heuristics still results in perfect recall, but with lower precision. All subkeys will still be extracted in stage 4, albeit mixed with other values. Filtering out the irrelevant values should not be too hard, however, if the encryption algorithm is known. The attacker can then check which potential subkeys form valid keys, and try those out on the data inputs.

We can thus conclude that the heuristics of stage 1 provide quite some redundancy, especially for simpler programs.

3.2 Stage 3

Our ablation study shows that K-Hunt++ achieves a perfect recall for any combination of two or more of the six stage 3 heuristics. In other words, up to four of the six stage 3 heuristics can be omitted and the approach will still rank the key-loading operations with top priority (assuming all stage 1 heuristics were deployed). The precision then goes down, however, as more heuristics being omitted results in more additional instructions being ranked with the same top priority as the key-loading instructions. Importantly, full precision is maintained if only one stage 3 heuristic is omitted, independent of which one is omitted, and for both use cases. Even for most combinations of two heuristics being omitted, full precision is obtained. For most combinations of three or four heuristics being omitted, the precision drops significantly, but not dramatically: only tens of additional instructions are ranked at the top priority, not hundreds, which implies that the attacker will still be able to filter out irrelevant subkeys reported by stage 4 with little effort.

It is clear that K-Hunt++ is much more robust than K-Hunt.

4 Robustness with Respect to Obfuscation

This section presents an analysis of the robustness of K-Hunt++, i.e., of its resilience against software protections that aim to break its heuristics. We structured this analysis per milepost.

First, however, we want to discuss one general protection strategy, namely where the defender injects additional code into the program with features and behavior similar to those of the mileposts. The software could, e.g., be transformed to execute multiple encryptions/decryptions/signatures on the input data, one with the true keys and additional ones with fake keys. While such additive attack strategies would result in more candidates for each milepost being identified, the number of fake and true (sub)keys being collected would in the end only increase linearly with the overhead that such a defense introduces. Some additional effort would hence be required by the attacker to sift out fake (sub)keys, but that effort would be limited. In the remainder of this section, we will hence neglect such additive protection strategies.

As for the use of dynamic analysis techniques, we refer to the report of the 2019 Dagstuhl seminar on Software Protection Decision Support and Evaluation Methodologies, which states that trace-based techniques have to be assumed possible given modern virtualization and instrumentation technologies, i.e., that they cannot be prevented entirely with software protections [10].

Finally, we note that the redundancy of the heuristics as observed in the ablation study does not imply that obfuscations that target those heuristics can have no effect on the recall of K-Hunt++. Omitting a heuristic as we did in the ablation study is equivalent to giving all candidate blocks, instructions, or operands *the same score* for the considered fingerprint. By contrast, obfuscations might be able to make non-key-loading instructions receive *higher scores* than key-loading ones, thus impacting both the precision and the recall of K-Hunt++ more than omitting heuristics does.

Milepost 1: crypto basic blocks. Heuristic (b) on the scaling of execution counts can obviously not be broken without severely impacting the user experience. Heuristic (d) on the high randomness of accessed data cannot be broken, as ciphertexts have high entropy by construction. Heuristic (c) on coverage can easily be broken, as it simply does not work in software in which the cryptographic functionality is compulsory. Heuristic (a) can be broken by injecting bogus crypto-like code into non-crypto functionality.

Milepost 2: key loading blocks. For expanding the set of basic blocks in which to find key loading instructions, we rely on data dependencies. One could try to thwart this with implicit data flow [3, 8, 28]. Such obfuscations introduce large overheads, however. Given that different subkey values have to be loaded repeatedly (at least on current architectures, with their current amounts of registers), in the inner loops of the encryption routines, such obfuscation will result in a huge slowdown. Alternatively, one can try to inject longer dependency chains of computational instructions spread over multiple basic blocks. That would introduce less overhead, but it can also be mitigated by our approach, namely by expanding the set of basic blocks identified in stage 2 iteratively. This will result in more code being instrumented for milepost 3, and hence will slow down the attack, but like the additive attacks mentioned above, we conjecture this can only result in a limited number of false positive results coming out of stages 3 and 4. Validating this is future work.

Milepost 3: key operands. For heuristic (e), at least three alternative analyses are available: fine-grained taint tracking like our updated version of Yadegari's technique, course-grained taint-tracking like that of the original K-Hunt, and K-Hunt++'s approach based on data dependency distances. While all of them can in theory be broken with implicit data flow obfuscations, we do not consider implicit data flow deployed on input data feasible: propagating all input data such as large documents implicitly through a program will simply be too slow to be acceptable for the user.

Alternatively, one can try to taint the key buffers in an idempotent manner, e.g., by XOR-ing the key data twice with data derived from the inputs. This would be relatively straightforward. We conjecture, however, that the distance-based approach will not be thwarted by this easily: the path from the data reading system call to the key will still be longer that the typically very short path to the encrypting XOR. Validating this is future work.

Even without obfuscations being deployed, heuristic (f) might not be applicable, as was already discussed for AES CBC mode, which uses small buffers of constant size for keys and for data, so data cannot be discriminated from keys based on the buffers' size and variability thereof. As the determination of the buffers' size and its variability is based on the addresses of the memory locations that instructions access during the execution, making the key buffers look larger and variable to make them similar to data buffers requires making the key-loading instructions access more memory locations and varying their amount based on input sizes. While custom obfuscations might be able to achieve this, we are not aware of any such obfuscations being described in literature.

With the exception of standardized constants that can be obfuscated such that they are not loaded from buffers but computed, heuristics (g)–(j) target rather fundamental properties that cannot be worked around without altering the core semantics of cryptographic standards. Heuristic (g) is broken when, e.g., salting is used, as was the case in our GPG use case, but (h) then provides a fallback. While this property might not hold for stream ciphers, we do think it targets a fundamental property of block ciphers.

In summary, while there certainly exist options to break some of the heuristics used in stage 3, they are far from straightforward. Most importantly, as we observed in the ablation study, breaking only one or two of them will typically not suffice. So we conjecture that breaking stage 3 of K-Hunt++ will require costly obfuscations, with a large, and potentially unacceptable, price on performance.

Milepost 4: key values. We used a debugger to extract the keys being used in a run of the program. This is highly efficient, as the use of a debugger introduces hardly any overhead in the program. Anti-debugging techniques can be used to prevent the use of a debugger. Anti-debugging checks that query the environment (e.g., via standard library APIs) for signs they are being debugged can easily be defeated as debuggers can intercept those queries [23]. A stronger form of anti-debugging is by means of self-debugging [1, 2, 12, 26]. While no successful attacks have been published on the strongest forms thereof, such as the circular self-debugging by Abrath et al. [1], those forms introduce quite some overhead, as well as additional complexity in the software development life cycle. Moreover, since the attacker aiming for milepost 4 already reached the previous milepost with dynamic techniques based on traces, they might just as well use similar dynamic analyses, implemented, e.g., with Pintools, to extract the keys.

5 Conclusions

We presented the K-Hunt++ crypto key extraction approach, an extension and improvement over K-Hunt. On two use cases we showed how K-Hunt++ correctly handles cases that K-Hunt can (likely) not handle. With an ablation study and a qualitative analysis, we provided evidence for the robustness of K-Hunt++. All its artifacts are available at https://github.com/csl-ugent/TREX.

Acknowledgments

This research was partly funded by the Cybersecurity Research Program Flanders. Thomas Faingnaert is supported with a PhD Fellowship grant from the Research Foundation - Flanders (FWO) Grant No. 11I1123N.

References

[1] Bert Abrath, Bart Coppens, Ilja Nevolin, and Bjorn De Sutter. 2020. Resilient self-debugging software protection. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 606–615.

[2] Bert Abrath, Bart Coppens, Stijn Volckaert, Joris Wijnant, and Bjorn De Sutter. 2016. Tightly-coupled self-debugging software protection. In *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*. 1–10.

[3] Golam Sarwar Babil, Olivier Mehani, Roksana Boreli, and Mohamed-Ali Kaafar. 2013. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *2013 International Conference on Security and Cryptography (SECRYPT)*. IEEE, 1–8.

[4] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX annual technical conference, FREENIX Track*, Vol. 41. California, USA, 10–5555.

[5] Léonard Benedetti, Aurélien Thierry, and Julien Francq. 2017. Detection of cryptographic algorithms with grap. *Cryptology ePrint Archive* 2017/1119 (2017). https://eprint.iacr.org/2017/1119

[6] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. 2009. Dispatcher: enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '09)*. Association for Computing Machinery, New York, NY, USA, 621–634. https://doi.org/10.1145/1653662.1653737

[7] Joan Calvet, José M Fernandez, and Jean-Yves Marion. 2012. Aligot: Cryptographic function identification in obfuscated binary programs. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 169–182.

[8] Lorenzo Cavallaro, Prateek Saxena, and R Sekar. 2007. Anti-taint-analysis: Practical evasion techniques against information flow based malware defense. *Secure Systems Lab at Stony Brook University, Tech. Rep* (2007), 1–18.

[9] B. R. Chandavarkar. 2020. Hardcoded Credentials and Insecure Data Transfer in IoT: National and International Status. In *2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. 1–7. https://doi.org/10.1109/ICCCNT49239.2020.9225520

[10] Bjorn De Sutter, Christian Collberg, Mila Dalla Preda, and Brecht Wyseur. 2019. Software Protection Decision Support and Evaluation Methodologies (Dagstuhl Seminar 19331). *Dagstuhl Reports* 9, 8 (2019), 1–25. https://doi.org/10.4230/DagRep.9.8.1

[11] Antonio M Espinoza, Jeffrey Knockel, Pedro Comesaña-Alfaro, and Jedidiah R Crandall. 2016. V-DIFT: Vector-based dynamic information flow tracking with application to locating cryptographic keys for reverse engineering. In *2016 11th International Conference on Availability, Reliability and Security (ARES)*. IEEE, 266–271.

[12] Peter Ferrie. 2008. Anti-Unpacker Tricks. In *CARO*.

[13] Hex-Rays. 2023. F.L.I.R.T: Fast Library Identification and Recognition Technology. https://hex-rays.com/products/ida/tech/flirt.

[14] Diane Duros Hosfelt. [n. d.]. *Automated detection and classification of cryptographic algorithms in binary programs through machine learning*. Master's thesis. Johns Hopkins University.

[15] Intel Corporation. 2023. Pin - A Dynamic Binary Instrumentation Tool. https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html.

[16] Burt Kaliski. 2000. *PKCS# 5: Password-based cryptography specification version 2.0.* Technical Report.

[17] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. 2014. Why does cryptographic software fail? a case study and open problems. In *Proceedings of 5th Asia-Pacific Workshop on Systems* (Beijing, China) *(APSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 7, 7 pages. https://doi.org/10.1145/2637166.2637237

[18] Pierre Lestringant, Frédéric Guihéry, and Pierre-Alain Fouque. 2015. Automated Identification of Cryptographic Primitives in Binary Code with Data Flow Graph Isomorphism. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (Singapore, Republic of Singapore) *(ASIA CCS '15)*. Association for Computing Machinery, New York, NY, USA, 203–214. https://doi.org/10.1145/2714576.2714639

[19] Juanru Li, Zhiqiang Lin, Juan Caballero, Yuanyuan Zhang, and Dawu Gu. 2018. K-Hunt: Pinpointing Insecure Cryptographic Keys from Execution Traces. In *Proc. 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018-10-15). 412–425. https://doi.org/10.1145/3243734.3243783

[20] Xin Li, Xinyuan Wang, and Wentao Chang. 2012. CipherXRay: Exposing cryptographic operations and transient secrets from monitored binary execution. *IEEE transactions on dependable and secure computing* 11, 2 (2012), 101–114.

[21] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.

[22] Carlo Meijer, Veelasha Moonsamy, and Jos Wetzels. 2021. Where's Crypto?: Automated Identification and Classification of Proprietary Cryptographic Primitives in Binary Code. In *30th USENIX Security Symposium (USENIX Security 21)*. 555–572.

[23] Henry Miller. 2005. Beginners Guide to Basic Linux Anti-Anti-Debugging Techniques. *CodeBreakers Journal* (2005).

[24] Jasvir Nagra and Christian Collberg. 2009. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Pearson Education.

[25] Igor Pavlov. 2023. 7-Zip. https://www.7-zip.org.

[26] Pellsson. 2010. Starcraft 2 Anti-Debugging. https://tinyurl.com/tyxjkeb

[27] The GnuPG Project. 2023. The GNU Privacy Guard. https://gnupg.org.

[28] Jon Stephens, Babak Yadegari, Christian Collberg, Saumya Debray, and Carlos Scheidegger. 2018. Probabilistic Obfuscation Through Covert Channels. In *2018 IEEE European Symposium on Security and Privacy*. 243–257. https://doi.org/10.1109/EuroSP.2018.00025

[29] Benjamin Taubmann, Omar Alabduljaleel, and Hans P Reiser. 2018. DroidKex: Fast extraction of ephemeral TLS keys from the memory of Android apps. *Digital Investigation* 26 (2018), S67–S76.

[30] The LLDB Team. 2023. The LLDB Debugger. https://lldb.llvm.org.

[31] Dongpeng Xu, Jiang Ming, and Dinghao Wu. 2017. Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 921–937.

[32] Babak Yadegari and Saumya Debray. 2014. Bit-Level Taint Analysis. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation* (Victoria, BC, Canada, 2014-09). IEEE, 255–264. https://doi.org/10.1109/SCAM.2014.43

[33] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. 2015. A Generic Approach to Automatic Deobfuscation of Executable Code. In *2015 IEEE Symposium on Security and Privacy*. 674–691. https://doi.org/10.1109/SP.2015.47

A Detailed results of the ablation study

Tables 1 and 2 report the qualitative results of our ablation study, on the basis of which Section 3 presented qualitative observations.

Table 1: Detailed results for the ablation study of stage 1. $|\text{out}_1|$ and $|\text{out}_2|$ are the amount of basic blocks that are output by stage 1 and stage 2, respectively. rank_3 lists the ranking of the key-loading instructions after stage 3 when all heuristics are used in that stage. 1–X indicates that along with the key-loading instructions, $X-16$ (for 7-Zip) and $X-3$ (for GPG) additional instructions are being ranked at that top priority. 1–16 (7-Zip) and 1–3 (GPG) indicate that the key-loading instructions got the top priority exclusively, resulting in full, precise key extraction, i.e., perfect recall and precision. The different parts of the table correspond to different amounts of heuristics being omitted from stage 1, ranging from no heuristics omitted in the top part, to three out of four heuristics omitted in the bottom part.

				7-Zip			GPG		
(a)	(b)	(c)	(d)	$\lvert\text{out}_1\rvert$	$\lvert\text{out}_2\rvert$	rank_3	$\lvert\text{out}_1\rvert$	$\lvert\text{out}_2\rvert$	rank_3
✓	✓	✓	✓	2	18	1–16	1	48	1–3
×	✓	✓	✓	3	22	1–16	1	48	1–3
✓	×	✓	✓	2	18	1–16	1	48	1–3
✓	✓	×	✓	8	19	1–16	4	107	1–7
✓	✓	✓	×	6	45	1–16	1	48	1–3
×	×	✓	✓	2	18	1–16	1	48	1–3
×	✓	×	✓	8	23	1–16	4	107	1–7
×	✓	✓	×	3	22	1–16	1	48	1–3
✓	×	×	✓	306	679	1–16	119	836	1–29
✓	×	✓	×	13	102	1–16	4	107	1–7
✓	✓	×	×	6	45	1–16	1	48	1–3
×	×	×	✓	899	1205	1–16	211	1094	1–32
×	×	✓	×	21	183	1–16	5	140	1–7
×	✓	×	×	3344	5684	1–16	3889	5728	1–110
✓	×	×	×	3344	5684	1–16	3889	5728	1–110

Table 2: Detailed results for the ablation study of stage 3. The 1–X in rank$_3$ has the same meaning as in Table 1. The different parts of the table correspond to different amounts of heuristics being omitted from stage 3, ranging from no heuristics omitted in the top part, to four out of six heuristics omitted in the bottom part.

						7-Zip	GPG
(e)	(f)	(g)	(h)	(i)	(j)	rank$_3$	rank$_3$
✓	✓	✓	✓	✓	✓	1–16	1–3
×	✓	✓	✓	✓	✓	1–16	1–3
✓	×	✓	✓	✓	✓	1–16	1–3
✓	✓	×	✓	✓	✓	1–16	1–3
✓	✓	✓	×	✓	✓	1–16	1–3
✓	✓	✓	✓	×	✓	1–16	1–3
✓	✓	✓	✓	✓	×	1–16	1–3
×	×	✓	✓	✓	✓	1–16	1–3
×	✓	×	✓	✓	✓	1–16	1–3
×	✓	✓	×	✓	✓	1–16	1–3
×	✓	✓	✓	×	✓	1–16	1–3
×	✓	✓	✓	✓	×	1–16	1–3
✓	×	×	✓	✓	✓	1–16	1–3
✓	×	✓	×	✓	✓	1–16	1–3
✓	×	✓	✓	×	✓	1–16	1–4
✓	×	✓	✓	✓	×	1–16	1–3
✓	✓	×	×	✓	✓	1–20	1–13
✓	✓	×	✓	×	✓	1–17	1–14
✓	✓	×	✓	✓	×	1–16	1–3
✓	✓	✓	×	×	✓	1–16	1–9
✓	✓	✓	×	✓	×	1–16	1–3
✓	✓	✓	✓	×	×	1–16	1–3
×	×	×	✓	✓	✓	1–16	1–3
×	×	✓	×	✓	✓	1–16	1–3
×	×	✓	✓	×	✓	1–16	1–4
×	×	✓	✓	✓	×	1–16	1–3
×	✓	×	×	✓	✓	1–26	1–15
×	✓	×	✓	×	✓	1–17	1–20
×	✓	×	✓	✓	×	1–16	1–3
×	✓	✓	×	×	✓	1–16	1–10
×	✓	✓	×	✓	×	1–16	1–3
×	✓	✓	✓	×	×	1–16	1–3
✓	×	×	×	✓	✓	1–20	1–27
✓	×	×	✓	×	✓	1–17	1–15
✓	×	×	✓	✓	×	1–16	1–3
✓	×	✓	×	×	✓	1–16	1–17
✓	×	✓	×	✓	×	1–16	1–8
✓	×	✓	✓	×	×	1–16	1–10
✓	✓	×	×	×	✓	1–23	1–56
✓	✓	×	×	✓	×	1–20	1–13
✓	✓	×	✓	×	×	1–20	1–14
✓	✓	✓	×	×	×	1–16	1–10
×	×	×	×	✓	✓	1–29	1–36
×	×	×	✓	×	✓	1–17	1–22
×	×	×	✓	✓	×	1–16	1–3
×	×	✓	×	×	✓	1–16	1–21
×	×	✓	×	✓	×	1–16	1–8
×	×	✓	✓	×	×	1–16	1–10
×	✓	×	×	×	✓	1–34	1–68
×	✓	×	×	✓	×	1–26	1–15
×	✓	×	✓	×	×	1–20	1–20
×	✓	✓	×	×	×	1–16	1–10
✓	×	×	×	×	✓	1–23	1–82
✓	×	×	×	✓	×	1–20	1–32
✓	×	×	✓	×	×	1–20	1–23
✓	×	✓	×	×	×	1–16	1–32
✓	✓	×	×	×	×	1–40	1–58